PREFACE

■ Objectives and targeted reader
FR81 Family is a 32 bit single chip microcontroller with CPU of new RISC Architecture as the core. FR81 Family has specifications that are optimum for embedded use requiring high performance CPU processing power.

This manual explains the programming model and execution instructions for engineers developing a product using this FR81 Family Microcontroller, especially the programmers who produce programs using assembly language of the assembler for FR/FR80/FR81 Family.

For the rules of assembly grammar language and the method of use of Assembler Programs, kindly refer to "FR Family Assembler Manual".

*: FR, the abbreviation of Fujitsu RISC controller, is a line of products of Fujitsu Microelectronics Limited.

Other company names and brand names are the trademarks or registered trademarks of their respective owners.

■ Organization of this Manual
This manual consists of the following 7 chapters and 1 supplement.

CHAPTER 1 OVERVIEW OF FR81 FAMILY CPU
This chapter describes the features of FR81 Family CPU and its differences from hitherto FR Family.

CHAPTER 2 MEMORY ARCHITECTURE
This chapter describes Memory Architecture of the CPU of FR81 Family. Memory Architecture is the method of allocation of memory space and access to this memory space.

CHAPTER 3 PROGRAMMING MODEL
This chapter describes registers in the CPU existing as programming model of FR81 Family CPU.

CHAPTER 4 RESET AND "EIT" PROCESSING
This chapter describes resetting of FR81 Family CPU and EIT processing. EIT processing is the generic term for exceptions, interruption and trap.

CHAPTER 5 PIPELINE OPERATION
This chapter describes pipeline operation and delay divergence, the salient feature of FR81 Family CPU.

CHAPTER 6 INSTRUCTION OVERVIEW
This chapter describes outline of commands of FR81 Family CPU.

CHAPTER 7 DETAILED EXECUTION INSTRUCTIONS
This chapter describes Execution Instructions of FR81 Family CPU in Reference Format in the alphabetical order.

APPENDIX
It contains instruction list and instruction map of FR81 Family CPU.
• The contents of this document are subject to change without notice. Customers are advised to consult with sales representatives before ordering.

• The information, such as descriptions of function and application circuit examples, in this document are presented solely for the purpose of reference to show examples of operations and uses of FUJITSU MICROELECTRONICS device; FUJITSU MICROELECTRONICS does not warrant proper operation of the device with respect to use based on such information. When you develop equipment incorporating the device based on such information, you must assume any responsibility arising out of such use of the information. FUJITSU MICROELECTRONICS assumes no liability for any damages whatsoever arising out of the use of the information.

• Any information in this document, including descriptions of function and schematic diagrams, shall not be construed as license of the use or exercise of any intellectual property right, such as patent right or copyright, or any other right of FUJITSU MICROELECTRONICS or any third party or does FUJITSU MICROELECTRONICS warrant non-infringement of any third-party's intellectual property right or other right by using such information. FUJITSU MICROELECTRONICS assumes no liability for any infringement of the intellectual property rights or other rights of third parties which would result from the use of information contained herein.

• The products described in this document are designed, developed and manufactured as contemplated for general use, including without limitation, ordinary industrial use, general office use, personal use, and household use, but are not designed, developed and manufactured as contemplated (1) for use accompanying fatal risks or dangers that, unless extremely high safety is secured, could have a serious effect to the public, and could lead directly to death, personal injury, severe physical damage or other loss (i.e., nuclear reaction control in nuclear facility, aircraft flight control, air traffic control, mass transport control, medical life support system, missile launch control in weapon system), or (2) for use requiring extremely high reliability (i.e., submersible repeater and artificial satellite).

Please note that FUJITSU MICROELECTRONICS will not be liable against you and/or any third party for any claims or damages arising in connection with above-mentioned uses of the products.

• Any semiconductor devices have an inherent chance of failure. You must protect against injury, damage or loss from such failures by incorporating safety design measures into your facility and equipment such as redundancy, fire protection, and prevention of over-current levels and other abnormal operating conditions.

• Exportation/release of any products described in this document may require necessary procedures in accordance with the regulations of the Foreign Exchange and Foreign Trade Control Law of Japan and/or US export control laws.

• The company names and brand names herein are the trademarks or registered trademarks of their respective owners.

Copyright ©2009 FUJITSU MICROELECTRONICS LIMITED All rights reserved.
## CONTENTS

### CHAPTER 1 OVERVIEW OF FR81 FAMILY CPU
1.1 Features of FR81 Family CPU .............................................................. 2
1.2 Changes from the earlier FR Family ..................................................... 4

### CHAPTER 2 MEMORY ARCHITECTURE
2.1 Address Space ......................................................................................... 8
  2.1.1 Direct Address Area ................................................................. 8
  2.1.2 Vector Table Area ....................................................................... 9
  2.1.3 20-bit Addressing Area & 32-bit Addressing Area ...................... 11
2.2 Data Structure ....................................................................................... 12
  2.2.1 Byte Data .................................................................................. 12
  2.2.2 Half Word Data .......................................................................... 12
  2.2.3 Word Data ................................................................................. 12
  2.2.4 Byte Order ................................................................................ 13
2.3 Word Alignment .................................................................................... 14
  2.3.1 Program Access ......................................................................... 14
  2.3.2 Data Access .............................................................................. 14

### CHAPTER 3 PROGRAMMING MODEL
3.1 Register Configuration ........................................................................... 16
3.2 General-purpose Registers .................................................................... 17
  3.2.1 Configuration of General-purpose Registers .............................. 17
  3.2.2 Special Usage of General-purpose Registers .............................. 18
  3.2.3 Relation between Stack Pointer and R15 ................................... 18
3.3 Dedicated Registers ............................................................................. 19
  3.3.1 Configuration of Dedicated Registers ....................................... 19
  3.3.2 Program Counter (PC) ............................................................... 20
  3.3.3 Program Status (PS) ................................................................. 20
  3.3.4 System Status Register (SSR) .................................................... 21
  3.3.5 Interrupt Level Mask Register (ILM) ........................................... 22
  3.3.6 Condition Code Register (CCR) ............................................... 23
  3.3.7 System Condition Code Register (SCR) ..................................... 25
  3.3.8 Return Pointer (RP) ................................................................. 26
  3.3.9 System Stack Pointer (SSP) ...................................................... 27
  3.3.10 User Stack Pointer (USP) .......................................................... 27
  3.3.11 Table Base Register (TBR) ....................................................... 29
  3.3.12 Multiplication/Division Register (MDH, MDL) ......................... 30
  3.3.13 Base Pointer (BP) ................................................................. 32
  3.3.14 FPU Control Register (FCR) .................................................... 32
  3.3.15 Exception status register (ESR) ................................................ 37
  3.3.16 Debug Register (DBR) ............................................................. 39
3.4 Floating-point Register ........................................................................ 40
## CHAPTER 4  RESET AND "EIT" PROCESSING ................................................................. 41
4.1  Reset .................................................................................................................. 42
4.2  Basic Operations in EIT Processing ................................................................. 43
  4.2.1  Types of EIT Processing and Prior Preparation ............................................ 43
  4.2.2  EIT Processing Sequence ........................................................................... 44
  4.2.3  Recovery from EIT Processing .................................................................... 45
4.3  Processor Operation Status ............................................................................. 46
4.4  Exception Processing ......................................................................................... 48
  4.4.1  Invalid Instruction Exception ..................................................................... 48
  4.4.2  Instruction Access Protection Violation Exception ........................................ 49
  4.4.3  Data Access Protection Violation Exception ................................................ 49
  4.4.4  FPU Exception ............................................................................................ 50
  4.4.5  Instruction Break ....................................................................................... 51
  4.4.6  Guarded Access Break ............................................................................... 52
4.5  Interrupts ........................................................................................................... 53
  4.5.1  General interrupts ....................................................................................... 53
  4.5.2  Non-maskable Interrupts (NMI) ................................................................. 55
  4.5.3  Break Interrupt ............................................................................................ 55
  4.5.4  Data Access Error Interrupt ........................................................................ 56
4.6  Traps .................................................................................................................... 57
  4.6.1  INT Instructions .......................................................................................... 57
  4.6.2  INTE Instruction ......................................................................................... 57
  4.6.3  Step Trace Traps ......................................................................................... 58
4.7  Multiple EIT processing and Priority Levels ....................................................... 60
  4.7.1  Multiple EIT Processing ............................................................................. 60
  4.7.2  Priority Levels of EIT Requests ................................................................. 61
  4.7.3  EIT Acceptance when Branching Instruction is Executed ............................ 62
4.8  Timing When Register Settings Are Reflected ..................................................... 63
  4.8.1  Timing when the interrupt enable flag (I) is requested .................................. 63
  4.8.2  Timing of Reflection of Interrupt Level Mask Register (ILM) ....................... 64
4.9  Usage Sequence of General Interrupts ............................................................... 65
  4.9.1  Preparation while using general interrupts .................................................. 65
  4.9.2  Processing during an Interrupt Processing Routine ....................................... 66
  4.9.3  Points of Caution while using General Interrupts ......................................... 66
4.10  Precautions ....................................................................................................... 67
  4.10.1  Exceptions in EIT Sequence and RETI Sequence ....................................... 67
  4.10.2  Exceptions in Multiple Load and Multiple Store Instructions ...................... 67
  4.10.3  Exceptions in Direct Address Transfer Instruction ....................................... 67

## CHAPTER 5  PIPELINE OPERATION .................................................................... 69
5.1  Instruction execution based on Pipeline ............................................................. 70
  5.1.1  Integer Pipeline .......................................................................................... 70
  5.1.2  Floating Point Pipeline .............................................................................. 72
5.2  Pipeline Operation and Interrupt Processing ..................................................... 73
  5.2.1  Mismatch in Acceptance and Cancellation of Interrupt ............................... 73
  5.2.2  Method of preventing the mismatched pipeline conditions ......................... 73
5.3  Pipeline hazards ................................................................................................. 74
7.15 ASR (Arithmetic shift to the Right Direction) ................................................................. 133
7.16 ASR2 (Arithmetic shift to the Right Direction) ............................................................. 135
7.17 BANDH (And 4bit Immediate Data to Higher 4bit of Byte Data in Memory) ............ 137
7.18 BANDL (And 4bit Immediate Data to Lower 4bit of Byte Data in Memory) ............ 139
7.19 Bcc (Branch relative if Condition satisfied) ................................................................. 141
7.20 Bcc:D (Branch relative if Condition satisfied) ............................................................. 143
7.21 BEORH (Eor 4bit Immediate Data to Higher 4bit of Byte Data in Memory) ............ 145
7.22 BEORL (Eor 4bit Immediate Data to Lower 4bit of Byte Data in Memory) ............ 147
7.23 BORH (Or 4bit Immediate Data to Higher 4bit of Byte Data in Memory) ............. 149
7.24 BORL (Or 4bit Immediate Data to Lower 4bit of Byte Data in Memory) ............. 151
7.25 BTSTH (Test Higher 4bit of Byte Data in Memory) .................................................... 153
7.26 BTSTL (Test Lower 4bit of Byte Data in Memory) ....................................................... 155
7.27 CALL (Call Subroutine) ............................................................................................... 157
7.28 CALL (Call Subroutine) ............................................................................................... 159
7.29 CALL:D (Call Subroutine) .......................................................................................... 161
7.30 CALL:D (Call Subroutine) .......................................................................................... 163
7.31 CMP (Compare Immediate Data and Destination Register) ...................................... 165
7.32 CMP (Compare Word Data in Source Register and Destination Register) .......... 167
7.33 CMP2 (Compare Immediate Data and Destination Register) ................................... 169
7.34 DIV0S (Initial Setting Up for Signed Division) ............................................................ 171
7.35 DIV0U (Initial Setting Up for Unsigned Division) ......................................................... 173
7.36 DIV1 (Main Process of Division) .................................................................................. 175
7.37 DIV2 (Correction When Remain is zero) ..................................................................... 177
7.38 DIV3 (Correction When Remain is zero) ..................................................................... 179
7.39 DIV4S (Correction Answer for Signed Division) ......................................................... 181
7.40 DMOV (Move Word Data from Direct Address to Register) ...................................... 183
7.41 DMOV (Move Word Data from Register to Direct Address) ..................................... 185
7.42 DMOV (Move Word Data from Direct Address to Post Increment Register Indirect Address) ...... 187
7.43 DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address) ...... 189
7.44 DMOV (Move Word Data from Direct Address to Pre Decrement Register Indirect Address) ...... 191
7.45 DMOVB (Move Word Data from Post Increment Register Indirect Address to Direct Address) ...... 193
7.46 DMOVB (Move Byte Data from Direct Address to Register) ....................................... 195
7.47 DMOVB (Move Byte Data from Register to Direct Address) ....................................... 197
7.48 DMOVB (Move Byte Data from Direct Address to Post Increment Register Indirect Address) ...... 199
7.49 DMOVB (Move Byte Data from Post Increment Register Indirect Address to Direct Address) ...... 201
7.50 DMOVH (Move Halfword Data from Direct Address to Register) ................................. 203
7.51 DMOVH (Move Halfword Data from Register to Direct Address) ................................. 205
7.52 DMOVH (Move Halfword Data from Direct Address to Post Increment Register Indirect Address) ...... 207
7.53 DMOVH (Move Halfword Data from Post Increment Register Indirect Address to Direct Address) ...... 209
7.54 ENTER (Enter Function) ............................................................................................. 211
7.55 EOR (Exclusive Or Word Data of Source Register to Data in Memory) .................... 213
7.56 EOR (Exclusive Or Word Data of Source Register to Destination Register) ............... 215
7.57 EORB (Exclusive Or Byte Data of Source Register to Data in Memory) ................... 217
7.58 EORH (Exclusive Or Halfword Data of Source Register to Data in Memory) .......... 219
7.59 EXTSB (Sign Extend from Byte Data to Word Data) .................................................... 221
7.60 EXTSH (Sign Extend from Byte Data to Word Data) .................................................... 223
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.61</td>
<td>EXTUB (Unsign Extend from Byte Data to Word Data)</td>
<td>225</td>
</tr>
<tr>
<td>7.62</td>
<td>EXTUH (Unsign Extend from Byte Data to Word Data)</td>
<td>227</td>
</tr>
<tr>
<td>7.63</td>
<td>FABSs (Single Precision Floating Point Absolute Value)</td>
<td>229</td>
</tr>
<tr>
<td>7.64</td>
<td>FADDS (Single Precision Floating Point Add)</td>
<td>230</td>
</tr>
<tr>
<td>7.65</td>
<td>FBcc (Floating Point Conditional Branch)</td>
<td>232</td>
</tr>
<tr>
<td>7.66</td>
<td>FBcc:D (Floating Point Conditional Branch with Delay Slot)</td>
<td>234</td>
</tr>
<tr>
<td>7.67</td>
<td>FCMPs (Single Precision Floating Point Compare)</td>
<td>236</td>
</tr>
<tr>
<td>7.68</td>
<td>FDIVs (Single Precision Floating Point Division)</td>
<td>238</td>
</tr>
<tr>
<td>7.69</td>
<td>FiTOs (Convert from Integer to Single Precision Floating Point)</td>
<td>240</td>
</tr>
<tr>
<td>7.70</td>
<td>FLD (Single Precision Floating Point Data Load)</td>
<td>242</td>
</tr>
<tr>
<td>7.71</td>
<td>FLD (Single Precision Floating Point Data Load)</td>
<td>243</td>
</tr>
<tr>
<td>7.72</td>
<td>FLD (Single Precision Floating Point Data Load)</td>
<td>244</td>
</tr>
<tr>
<td>7.73</td>
<td>FLD (Single Precision Floating Point Data Load)</td>
<td>245</td>
</tr>
<tr>
<td>7.74</td>
<td>FLD (Single Precision Floating Point Data Load)</td>
<td>246</td>
</tr>
<tr>
<td>7.75</td>
<td>FLD (Load Word Data in Memory to Floating Register)</td>
<td>247</td>
</tr>
<tr>
<td>7.76</td>
<td>FLD (Load Word Data in Memory to Register)</td>
<td>249</td>
</tr>
<tr>
<td>7.77</td>
<td>FMULs (Single Precision Floating Point Multiply)</td>
<td>250</td>
</tr>
<tr>
<td>7.78</td>
<td>FMOVs (Single Precision Floating Point Move)</td>
<td>252</td>
</tr>
<tr>
<td>7.79</td>
<td>FMSUBs (Single Precision Floating Point Multiply and Subtract)</td>
<td>253</td>
</tr>
<tr>
<td>7.80</td>
<td>FST (Single Precision Floating Point Data Store)</td>
<td>255</td>
</tr>
<tr>
<td>7.81</td>
<td>FNEG (Single Precision Floating Point sign reverse)</td>
<td>257</td>
</tr>
<tr>
<td>7.82</td>
<td>FSQRTs (Single Precision Floating Point Square Root)</td>
<td>258</td>
</tr>
<tr>
<td>7.83</td>
<td>FST (Single Precision Floating Point Data Store)</td>
<td>259</td>
</tr>
<tr>
<td>7.84</td>
<td>FST (Single Precision Floating Point Data Store)</td>
<td>260</td>
</tr>
<tr>
<td>7.85</td>
<td>FST (Single Precision Floating Point Data Store)</td>
<td>261</td>
</tr>
<tr>
<td>7.86</td>
<td>FST (Single Precision Floating Point Data Store)</td>
<td>262</td>
</tr>
<tr>
<td>7.87</td>
<td>FST (Single Precision Floating Point Data Store)</td>
<td>263</td>
</tr>
<tr>
<td>7.88</td>
<td>FST (Store Word Data in Floating Point Register to Memory)</td>
<td>264</td>
</tr>
<tr>
<td>7.89</td>
<td>FSTM (Single Precision Floating Point Data Store from Multiple Register)</td>
<td>265</td>
</tr>
<tr>
<td>7.90</td>
<td>FS TOI (Convert from Single Precision Floating Point to Integer)</td>
<td>267</td>
</tr>
<tr>
<td>7.91</td>
<td>FSUBs (Single Precision Floating Point Subtract)</td>
<td>269</td>
</tr>
<tr>
<td>7.92</td>
<td>INT (Software Interrupt)</td>
<td>271</td>
</tr>
<tr>
<td>7.93</td>
<td>INT (Software Interrupt)</td>
<td>273</td>
</tr>
<tr>
<td>7.94</td>
<td>JMP (Jump)</td>
<td>275</td>
</tr>
<tr>
<td>7.95</td>
<td>JMP: D (Jump)</td>
<td>277</td>
</tr>
<tr>
<td>7.96</td>
<td>LCALL (Long Call Subroutine)</td>
<td>279</td>
</tr>
<tr>
<td>7.97</td>
<td>LCALL: D (Long Call Subroutine)</td>
<td>280</td>
</tr>
<tr>
<td>7.98</td>
<td>LD (Load Word Data in Memory to Register)</td>
<td>281</td>
</tr>
<tr>
<td>7.99</td>
<td>LD (Load Word Data in Memory to Register)</td>
<td>283</td>
</tr>
<tr>
<td>7.100</td>
<td>LD (Load Word Data in Memory to Register)</td>
<td>285</td>
</tr>
<tr>
<td>7.101</td>
<td>LD (Load Word Data in Memory to Register)</td>
<td>287</td>
</tr>
<tr>
<td>7.102</td>
<td>LD (Load Word Data in Memory to Register)</td>
<td>289</td>
</tr>
<tr>
<td>7.103</td>
<td>LD (Load Word Data in Memory to Register)</td>
<td>291</td>
</tr>
<tr>
<td>7.104</td>
<td>LD (Load Word Data in Memory to Register)</td>
<td>292</td>
</tr>
<tr>
<td>7.105</td>
<td>LD (Load Word Data in Memory to Program Status Register)</td>
<td>294</td>
</tr>
<tr>
<td>7.106</td>
<td>LD:20 (Load Immediate 20bit Data to Destination Register)</td>
<td>296</td>
</tr>
<tr>
<td>7.107</td>
<td>LD:32 (Load Immediate 32 bit Data to Destination Register)</td>
<td>298</td>
</tr>
</tbody>
</table>
7.108 LDI:8 (Load Immediate 8bit Data to Destination Register) ................................................................. 300
7.109 LDM0 (Load Multiple Registers) ..................................................................................................... 302
7.110 LDM1 (Load Multiple Registers) ..................................................................................................... 304
7.111 LDUB (Load Byte Data in Memory to Register) ............................................................................... 306
7.112 LDUB (Load Byte Data in Memory to Register) ............................................................................... 308
7.113 LDUB (Load Byte Data in Memory to Register) ............................................................................... 310
7.114 LDUB (Load Byte Data in Memory to Register) ............................................................................... 312
7.115 LDUH (Load Halfword Data in Memory to Register) ...................................................................... 313
7.116 LDUH (Load Halfword Data in Memory to Register) ...................................................................... 315
7.117 LDUH (Load Halfword Data in Memory to Register) ...................................................................... 317
7.118 LDUH (Load Halfword Data in Memory to Register) ...................................................................... 319
7.119 LEAVE (Leave Function) .................................................................................................................. 320
7.120 LSL (Logical Shift to the Left Direction) ............................................................................................ 322
7.121 LSL (Logical Shift to the Left Direction) ............................................................................................ 324
7.122 LSL2 (Logical Shift to the Left Direction) .......................................................................................... 326
7.123 LSR (Logical Shift to the Right Direction) ....................................................................................... 328
7.124 LSR (Logical Shift to the Right Direction) ....................................................................................... 330
7.125 LSR2 (Logical Shift to the Right Direction) ..................................................................................... 332
7.126 MOV (Move Word Data in Source Register to Destination Register) ............................................. 334
7.127 MOV (Move Word Data in Source Register to Destination Register) ............................................. 336
7.128 MOV (Move Word Data in Program Status Register to Destination Register) ................................ 338
7.129 MOV (Move Word Data in Source Register to Destination Register) ............................................. 340
7.130 MOV (Move Word Data in Source Register to Program Status Register) ...................................... 342
7.131 MOV (Move Word Data in General Purpose Register to Floating Point Register) ....................... 344
7.132 MOV (Move Word Data in Floating Point Register to General Purpose Register) ..................... 345
7.133 MUL (Multiply Word Data) ............................................................................................................. 346
7.134 MULH (Multiply Halfword Data) ....................................................................................................... 348
7.135 MULU (Multiply Unsigned Word Data) ............................................................................................ 350
7.136 MULUH (Multiply Unsigned Halfword Data) .................................................................................... 352
7.137 NOP (No Operation) ........................................................................................................................ 354
7.138 OR (Or Word Data of Source Register to Data in Memory) ............................................................ 356
7.139 OR (Or Word Data of Source Register to Destination Register) .................................................... 358
7.140 ORB (Or Byte Data of Source Register to Data in Memory) ............................................................ 360
7.141 ORCCR (Or Condition Code Register and Immediate Data) .......................................................... 362
7.142 ORH (Or Halfword Data of Source Register to Data in Memory) .................................................... 364
7.143 RET (Return from Subroutine) .......................................................................................................... 366
7.144 RET:D (Return from Subroutine) ..................................................................................................... 368
7.145 RETI (Return from Interrupt) ........................................................................................................... 370
7.146 SRCCH0 (Search First Zero bit position distance From MSB) ............................................................ 373
7.147 SRCCH1 (Search First One bit position distance From MSB) ........................................................... 375
7.148 SRCHC (Search First bit value change position distance From MSB) ............................................. 377
7.149 ST (Store Word Data in Register to Memory) .................................................................................... 379
7.150 ST (Store Word Data in Register to Memory) .................................................................................... 381
7.151 ST (Store Word Data in Register to Memory) .................................................................................... 383
7.152 ST (Store Word Data in Register to Memory) .................................................................................... 385
7.153 ST (Store Word Data in Register to Memory) .................................................................................... 387
7.154 ST (Store Word Data in Register to Memory) .................................................................................... 389
CHAPTER 1

OVERVIEW OF FR81 FAMILY CPU

This chapter describes the features of FR81 Family CPU and the changes from the earlier FR Family.

1.1 Features of FR81 Family CPU
1.2 Changes from the earlier FR Family
Chapter 1  Overview of FR81 Family CPU

1.1 Features of FR81 Family CPU

FR81 Family CPU is meant for 32 bit RISC controller having proprietary FR81 architecture of Fujitsu. The FR81 architecture is optimized for microcontrollers by using the FR family instruction set and including improved floating-point, memory protection, and debug functions.

■ General-purpose Register Architecture

It is load/store architecture based on 16 numbers of 32-bit General-purpose registers R0 to R15. The architecture also has instructions that are suitable for embedded uses such as memory to memory transfer, bit processing etc.

■ Linear Space for 32-bit (4G bytes) addressing

Address space is controlled for each byte unit. Linear specification of Address is made based on 32-bit address.

■ 16-bit fixed instruction length (excluding immediate data transfer instructions)

It is 16-bit fixed length instruction format excluding 32/20-bit immediate data transfer instruction. It enables securing high object efficiency.

■ Floating point calculation unit (FPU)

FR81 Family supports single precision floating point calculation (IEEE754 compliant). It has 16 pieces of 32-bit floating point registers from FR0 to FR15. A single instruction can execute a product-sum operation type calculation (multiplication, or addition/subtraction). The instruction length of a floating point type instruction is 32 bits.

■ Pipeline Configuration

High speed one-instruction one-cycle processing of the basic instructions based on 5-stage pipeline operation can be carried out. Pipeline has following 5-stage configuration.

- IF Stage: Load Instruction
- ID Stage: Interpret Instruction
- EX Stage: Execute Instruction
- MA Stage: Memory Access
- WB Stage: Write to register

FR81 Family has the 6-stage pipeline configuration to execute floating point type instructions.

■ Non-blocking load

In FR81 Family, non-blocking loading is carried out making execution of LD (load) instructions efficient. A maximum of four LD (Load) instructions can be issued in anticipation. In non-blocking, succeeding instruction is executed without waiting for the completion of a load instruction, in case general-purpose register storing the value of load instruction is not referred by the succeeding instruction.
FR81 Family

■ Harvard Architecture
An instruction can be executed efficiently based on Harvard Architecture where instruction bus for instruction access and data bus for data access are independent.

■ Multiplication Instruction
Multiplication/division computation can be executed at the instruction level based on an in-built multiplier. 32-bit multiplication, signed or unsigned, is executed in 5 cycles. 16-bit multiplication is executed in 3 cycles.

■ Step Division Instruction
32-bit ÷ 32-bit division, signed or unsigned, can be executed based on combination of step division instructions.

■ Direct Addressing Instruction for peripheral access
Address of 256 words/256 half-words/256 bytes from the top of address space (low order address) can be directly specified. It is convenient for address specification in the I/O Register of the peripheral resource.

■ High-speed interrupt processing complete within 6 cycles
Acceptance of interruption is processed at a high speed within 6 cycles. A 16-level priority order is given to the request for interruption. Masking in line with the priority order can be carried out based on interruption mask level of the CPU.
1.2 Changes from the earlier FR Family

FR81 Family has partial addition and deletion of instructions and operational changes from the earlier FR Family (FR30 Family, FR60 Family etc.).

- **Instructions that cannot be used in FR81/FR80 Family**
  Following instructions cannot be used in FR81/FR80 Family.
  - Coprocessor Instructions (COPOP, COPLD, COPST, COPSV)
  - Resource Instructions (LDRES, STRES)
  Undefined Instruction Exceptions and not the Coprocessor Error Trap occur when execution of Coprocessor Instruction is attempted. Undefined Instruction Exceptions occur when execution of Resource Instruction is attempted.

- **Instructions added to FR81/FR80 Family**
  Following instructions have been added in FR81/FR80 Family. These instructions have replaced the bit search module embedded as a peripheral function.
  - SRCH1 (Bit Search Instruction Detection of First "1" bit from MSB to LSB)
  - SRCH0 (Bit Search Instruction Detection of first "0" bit from MSB to LSB)
  - SRCHC (Bit Search Instruction Detection of Change point from MSB to LSB)
  see "Chapter 7 Detailed Execution Instructions" and "Appendix A 2 Instruction Lists" for operation of Bit Search Instructions.

- **Adding floating point type instructions**
  Floating point type instructions and 16 pieces of 32-bit floating point registers (FR0 to FR15) have been added in FR81 Family.

- **Privilege mode**
  Privilege mode has been added in FR81 family. Privilege mode and user mode are two CPU operation modes.

- **Exception processing**
  Exception processing has been improved for FR81 Family. The following exceptions have been added.
  - FPU exception
  - Instruction access protection violation exception
  - Data access protection violation exception
  - Invalid instruction exception (Changing definition from undefined instruction exception)
  - Data access error exception
  - FPU absence exception
FR81 Family

■ Operation of INTE Instructions during Step Execution

In FR81 Family, trap processing is initiated based on INTE instructions even during step execution based on step trace trap.

In hitherto FR Family, trap processing is not initiated based on INTE instructions during step execution.

For trap processing based on step trace trap and INTE instructions, see “4.6 Traps”.
This chapter explains the memory architecture of FR81 Family CPU. Memory architecture refers to allocation of memory spaces and methods used to access memory.

2.1 Address Space
2.2 Data Structure
2.3 Word Alignment
2.1 Address Space

The address space of FR81 Family CPU is 32 bits (4Gbyte).

CPU controls the address spaces in byte units. An address on the address space is accessed from the CPU by specifying a 32-bit value. Address space is indicated in Figure 2.1-1.

Address space is also called memory space. It is a logical address space as seen from the CPU. Addresses cannot be changed. Logical address as seen from the CPU, and the physical address actually allocated to memory or I/O are identical.

2.1.1 Direct Address Area

In the lower address in the address space, there is a direct address area.

Direct address area directly specifies an address in the direct address specification instruction. This area accesses only based on operand data in the instruction without the use of general-purpose registers. The size of the address area that can be specified by direct addressing varies according to the data type being accessed.

The correspondence between data type and area specified by direct address is as follows.

- byte data access: 0000 0000H to 0000 00FFH
- half-word data access: 0000 0000H to 0000 01FFH
- word data access: 0000 0000H to 0000 03FFH

The method of using the 8-bit address data contained in the operand of instructions that specify direct addresses is as follows:
2.1 Byte data access: Lower 8 bits of the address are used as is  
2.1 Half word data access: Value is doubled and used as lower 9 bits of the address  
2.1 Word data access: Value is quadrupled and used as lower 10 bits of the address

The relation between data types specified by direct address and memory address is shown in Figure 2.1-2.

### Figure 2.1-2 Relation between data type specified by direct address and memory address

[Example 1] Byte data: DMOVB R13, @58H  
Object code: 1A58H  
Memory space

[Example 2] Half-word data: DMOVB R13, @58H  
Object code: 1B2CH  
Memory space

[Example 3] Word data: DMOVB R13, @58H  
Object code: 1B16H  
Memory space

### 2.1.2 Vector Table Area

An area of 1Kbyte from the address shown in the Table Base Register (TBR) is called the EIT Vector Table Area.

Table Base Register (TBR) represents the top address of the vector table area. In this vector table area, the entry addresses of EIT processing (Exception processing, Interrupt processing, Trap processing) are described. The relation between Table Base Register (TBR) and vector table area is shown in Figure 2.1-3.

### Figure 2.1-3 Relation between Table Base Register (TBR) and Vector Table Area addresses

<table>
<thead>
<tr>
<th>Number</th>
<th>Offset from TBR</th>
<th>EIT source</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFH</td>
<td>000H</td>
<td>Entry address for INT instruction</td>
</tr>
<tr>
<td>FEH</td>
<td>004H</td>
<td>Entry address for INT instruction</td>
</tr>
<tr>
<td>FDH</td>
<td>008H</td>
<td>Entry address for INT instruction</td>
</tr>
<tr>
<td>FCH</td>
<td>00CH</td>
<td>Entry address for INT instruction</td>
</tr>
<tr>
<td>00H</td>
<td>3FCH</td>
<td>Entry address for reset processing</td>
</tr>
</tbody>
</table>
CHAPTER 2  MEMORY ARCHITECTURE
FR81 Family

2.1

As a result of reset, the value of Table Base Register (TBR) is initialized to 000F FC00_H, and the range of vector table area extends from 000F FC00_H to 000F FFFF_H. By rewriting the Table Base Register (TBR), the vector table area can be allocated to any desired location.

A vector table is composed of entry addresses for each EIT processing programs. Each vector table contains values whose use is fixed according to the CPU architecture, and values that vary according to the type of built-in peripheral functions. The structure of vector table area is shown in Table 2.1-1.

Table 2.1-1  Structure of Vector Table Area

<table>
<thead>
<tr>
<th>Offset from TBR</th>
<th>Vector number</th>
<th>Model-dependence</th>
<th>EIT value description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>3FC_H</td>
<td>00_H</td>
<td>No</td>
<td>reset</td>
<td></td>
</tr>
<tr>
<td>3F8_H</td>
<td>01_H</td>
<td>No</td>
<td>system reserved</td>
<td></td>
</tr>
<tr>
<td>3F4_H</td>
<td>02_H</td>
<td>No</td>
<td>system reserved</td>
<td>Disabled</td>
</tr>
<tr>
<td>3F0_H</td>
<td>03_H</td>
<td>No</td>
<td>system reserved</td>
<td>Disabled</td>
</tr>
<tr>
<td>3E8_H</td>
<td>05_H</td>
<td>No</td>
<td>system reserved</td>
<td>Disabled</td>
</tr>
<tr>
<td>3E4_H</td>
<td>06_H</td>
<td>No</td>
<td>Instruction access protection violation exception</td>
<td></td>
</tr>
<tr>
<td>3E0_H</td>
<td>07_H</td>
<td>No</td>
<td>Data access protection violation exception</td>
<td></td>
</tr>
<tr>
<td>3DC_H</td>
<td>08_H</td>
<td>No</td>
<td>Data access error interrupt</td>
<td></td>
</tr>
<tr>
<td>3D8_H</td>
<td>09_H</td>
<td>No</td>
<td>INTE instruction</td>
<td>For use in the emulator</td>
</tr>
<tr>
<td>3D4_H</td>
<td>0A_H</td>
<td>No</td>
<td>Instruction break</td>
<td></td>
</tr>
<tr>
<td>3D0_H</td>
<td>0B_H</td>
<td>No</td>
<td>system reserved</td>
<td></td>
</tr>
<tr>
<td>3CC_H</td>
<td>0C_H</td>
<td>No</td>
<td>Step trace trap</td>
<td></td>
</tr>
<tr>
<td>3C8_H</td>
<td>0D_H</td>
<td>No</td>
<td>system reserved</td>
<td></td>
</tr>
<tr>
<td>3C4_H</td>
<td>0E_H</td>
<td>No</td>
<td>Invalid instruction exception</td>
<td></td>
</tr>
<tr>
<td>3C0_H</td>
<td>0F_H</td>
<td>No</td>
<td>NMI request</td>
<td></td>
</tr>
<tr>
<td>3BC_H to 304_H</td>
<td>0F_H to 3E_H</td>
<td>Yes</td>
<td>General interrupt (used in external interrupt, interrupt from peripheral function)</td>
<td>Refer to the Hardware Manual for each model</td>
</tr>
<tr>
<td>300_H</td>
<td>3F_H</td>
<td>No</td>
<td>General interrupts</td>
<td>Used in Delayed interrupt</td>
</tr>
<tr>
<td>2FC_H</td>
<td>40_H</td>
<td>No</td>
<td>system reserved</td>
<td>Used in REALOS</td>
</tr>
<tr>
<td>2F8_H</td>
<td>41_H</td>
<td>No</td>
<td>system reserved</td>
<td>Used in REALOS</td>
</tr>
<tr>
<td>2F4_H to 000_H</td>
<td>42_H to FF_H</td>
<td>No</td>
<td>Used in INT instruction</td>
<td></td>
</tr>
</tbody>
</table>

For vector tables of actual models, refer to the hardware manuals for each model.
2.1.3 20-bit Addressing Area & 32-bit Addressing Area

The lower portion of the address space extending from 0000 0000_H to 000F FFFF_H (1Mbyte) will be the 20-bit addressing area. The overall address space from 0000 0000_H to FFFF FFFF_H will be 32-bit addressing space.

If all the program locations and data locations are positioned within the 20-bit addressing area, a compact and high-speed program can be realized as compared to a 32-bit addressing area.

In a 20-bit addressing area, as the address values are within 20 bits, the LDI:20 instruction can be used for immediate loading of address information. The instruction length (Code size) of LDI:20 instruction is 4 bytes. By using LDI:20 instruction, the program becomes more compact than when using LDI:32 instruction of instruction length 6 bytes.

Example of 20-bit Addressing

<table>
<thead>
<tr>
<th>Code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDI:20 #label20,Ri</td>
</tr>
<tr>
<td>JMP @Ri</td>
</tr>
<tr>
<td>Total 6 bytes</td>
</tr>
</tbody>
</table>

Example of 32-bit Addressing

<table>
<thead>
<tr>
<th>Code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDI:32 #label32,Ri</td>
</tr>
<tr>
<td>JMP @Ri</td>
</tr>
<tr>
<td>Total 8 bytes</td>
</tr>
</tbody>
</table>
FR81 Family CPU has three data types namely byte data (8-bits), half word data (16-bits) and word data (32-bits). The byte order is big endian.

### 2.2 Data Structure

#### 2.2.1 Byte Data

This is a data type having 8 bits as unit. Bit order is little endian, MSB side becomes bit7 and LSB side becomes bit0. The structure of byte data is shown in Figure 2.2-1.

![Figure 2.2-1 Structure of byte data](image1)

#### 2.2.2 Half Word Data

This is a data type having 16 bits (2 byte) as unit. Bit order is little endian, MSB side is bit15 while LSB side is bit0. Bit15 to bit8 of MSB side represent the higher bytes while bit7 to bit0 of LSB side represent the lower bytes. The structure of half word data is shown in Figure 2.2-2.

![Figure 2.2-2 Structure of Half Word Data](image2)

#### 2.2.3 Word Data

This is a data type having 32 bits (4 byte) as unit. Bit order is little endian, MSB side is bit31 while LSB side is bit0. Bit31 to bit16 of the MSB side become the higher half word, while bit15 to bit0 of the LSB side become the lower half word. The structure of word data is shown in Figure 2.2-3.

![Figure 2.2-3 Structure of Word Data](image3)
2.2.4 Byte Order

The byte order of FR81 Family CPU is big endian. When word data or half word data are allocated to address spaces, the higher bytes are placed in the lower address side while the lower bytes are placed in the higher address side. The arrangement of big endian byte data is shown in Figure 2.2-4.

For example, if a word data was written on the memory (RAM) at address location 0004 1234H of the memory space, the highest byte will be stored at location 0004 1234H while the lowest byte will be stored at location byte 0004 1237H.

Figure 2.2-4 Big Endian Byte Order
2.3 Word Alignment

The data type used determines restrictions on the designation of memory addresses (word alignment).

2.3.1 Program Access

Unit of instruction length is half word (2byte) and all instructions are allocated to addresses which are multiples of 2 (2n location).

At the time of execution of the instruction, bit0 of the program counter (PC) automatically becomes "0", and is always at an even address. In a branched instruction, even if an odd address is generated as a result of branch destination address calculation, the bit0 of the address will be assigned "0" and branched to an even address.

There is no address exception in program access.

2.3.2 Data Access

There are following restrictions on addresses for data access depending upon the data type used.

Word data

Data is assigned to addresses that are multiples of 4 (4n location). The restriction of multiples of 4 on addresses is called ‘word boundary’. If the specified address is not a multiple of 4, the lower two bits of the address are set to "00" forcibly.

Half-word data

Data is assigned to addresses that are multiples of 2 (2n locations). The restriction of multiples of 2 on addresses is called ‘half-word boundary’. If the specified address is not a multiple of 2, the lower 1 bit of the address is set to "0" forcibly.

Byte data

There is no restriction on allocation of addresses.

During word and half-word data access, condition that lower bit of an address has to be "0" is applicable only for the result of computation of an effective address. Values still under calculation are used as they are.
This chapter describes the programming model of FR81 Family CPU.

3.1 Register Configuration
3.2 General-purpose Registers
3.3 Dedicated Registers
3.4 Floating-point Register
3.1 Register Configuration

FR81 Family CPU uses three types of registers, namely, general-purpose registers, dedicated registers and floating point registers.

General-purpose registers are registers that store computation data and address information. They comprise 16 registers from R0 to R15. Dedicated registers are registers that store information for specific applications.

Floating point registers are registers that store calculation information for floating point calculations. They are comprised of 16 registers from FR0 to FR15.
3.2 General-purpose Registers

General-purpose registers are used for storing results of various calculations, as well as information about addresses to be used as pointers for memory access.

3.2.1 Configuration of General-purpose Registers

General-purpose registers has sixteen each 32 bits in length. General-purpose registers have names R0 to R15.

In case of general instructions, the general-purpose registers can use without any distinction. In some instructions, three registers namely R13, R14 and R15 have special usages.

Figure 3.2-1 shows the configuration and initial values of general-purpose registers.

Figure 3.2-1 Configuration and initial values of general-purpose registers

R0 to R14 are not initialized as a result of reset. R15 is initialized 0000 0000H as a result of reset.
CHAPTER 3  PROGRAMMING MODEL

3.2  FR81 Family

3.2.2  Special Usage of General-purpose Registers

General-purpose registers R13 to R15, besides being used as other general-purpose registers, are used in the following way in some instructions.

R13 (Virtual Accumulator: AC)
- Base address register for load/store to memory instructions
  [Example: LD @(R13,Rj), Ri]
- Accumulator for direct address designation
  [Example: DMOV @dir10, R13]
- Memory pointer for direct address designation
  [Example: DMOV @dir10,@R13+]

R14 (Frame Pointer: FP)
- Index register for load/store to memory instructions
  [Example: LD @(R14,disp10), Ri]
- Frame pointer for reserve/release of dynamic memory area
  [Example: ENTER #u10]

R15 (Stack Pointer: SP)
- Index register for load/store to memory instructions
  [Example: LD @(R15,udisp6), Ri]
- Stack pointer
  [Example: LD @(R15+,Ri]
- Stack pointer for reserve/release of dynamic memory area
  [Example: ENTER #u10]

3.2.3  Relation between Stack Pointer and R15

R15 functions as an indirect register. Physically it becomes either the system stack pointer (SSP) or user pointer (USP) for dedicated registers. When the notation R15 is used in an instruction, this register will function as USP if the stack flag (S) is "1" and as SSP if the stack flag is "0". Table 3.2-1 shows the correlation between general-purpose register R15 and stack pointer.

When something is written on R15 as a general-purpose register, it is automatically written onto the system stack pointer (SSP) or user stack pointer (USP) according to the value of stack flag (S).

<table>
<thead>
<tr>
<th>General-purpose register</th>
<th>S Flag</th>
<th>Stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>R15</td>
<td>1</td>
<td>User stack pointer (USP)</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>System stack pointer (SSP)</td>
</tr>
</tbody>
</table>

Table 3.2-1  Correlation between General-purpose Register "R15" and Stack Pointer

Stack flag (S) is present in the condition code register (CCR) section of the program status (PS).
FR81 Family

3.3 Dedicated Registers

FR81 Family CPU has dedicated registers reserved for special usages.

3.3.1 Configuration of Dedicated Registers

Dedicated registers are used for special purposes. The following dedicated registers are available.

- Program counter (PC)
- Program status (PS)
- Return pointer (RP)
- System stack pointer (SSP)
- User stack pointer (USP)
- Table base register (TBR)
- Multiplication/Division Register (MDH, MDL)
- Base Pointer (BP)
- FPU control register (FCR)
- Exception status register (ESR)
- Debug register (DBR)

Figure 3.3-1 shows the configuration and initial values of dedicated registers.

**Figure 3.3-1 Configuration and Initial Values of Dedicated Registers**

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program counter (PC)</td>
<td>XXXX XXXX&lt;sub&gt;a&lt;/sub&gt;</td>
</tr>
<tr>
<td>Program status (PS)</td>
<td>ILM</td>
</tr>
<tr>
<td>Table base register (TBR)</td>
<td>0000</td>
</tr>
<tr>
<td>Return pointer (RP)</td>
<td>XXXX XXXX&lt;sub&gt;a&lt;/sub&gt;</td>
</tr>
<tr>
<td>System stack pointer (SSP)</td>
<td>0000</td>
</tr>
<tr>
<td>User stack pointer (USP)</td>
<td>XXXX XXXX&lt;sub&gt;a&lt;/sub&gt;</td>
</tr>
<tr>
<td>Multiplication/Division registers</td>
<td>XXXX XXXX&lt;sub&gt;a&lt;/sub&gt;</td>
</tr>
<tr>
<td>Multiplication/Division registers</td>
<td>XXXX XXXX&lt;sub&gt;a&lt;/sub&gt;</td>
</tr>
<tr>
<td>Base Pointer (BP)</td>
<td>XXXX XXXX&lt;sub&gt;a&lt;/sub&gt;</td>
</tr>
<tr>
<td>FPU control register (FCR)</td>
<td>XXXX XXXX&lt;sub&gt;a&lt;/sub&gt;</td>
</tr>
<tr>
<td>Exception status register (ESR)</td>
<td>0000</td>
</tr>
<tr>
<td>Debug register (DBR)</td>
<td>XXXX XXXX&lt;sub&gt;a&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
3.3.2 Program Counter (PC)

Program counter (PC) is a 32-bit register that indicates the address containing the instruction that is currently executing.

Figure 3.3-2 shows the bit configuration of program counter (PC).

![Figure 3.3-2 Program Counter (PC) Bit Configuration]

The value of the lowest bit (LBS) of the program counter (PC) is always read as “0”. Even if "1" is written to it as a result of address calculation of branching destination, the lowest bit of branching address will be treated as "0". When the program counter (PC) changes after the execution of an instruction and it indicates the next instruction, the lowest bit is always read as "0".

Following a reset, the contents of the Program Counter (PC) are set to the value (reset entry address) written in the reset vector of the vector table. As the table base register (TBR) is initialized first by reset, the address of the reset vector will be 000F FFFCh.

3.3.3 Program Status (PS)

Program status (PS) is a 32-bit register that indicates the status of program execution. It sets the interrupt enable level, controls the program trace break function in the CPU, and indicates the status of instruction execution.

Program status (PS) consists of the following 4 parts.

- System status register (SSR)
- Interrupt level mask register (ILM)
- System condition code register (SCR)
- Condition code register (CCR)

Figure 3.3-3 shows the bit configuration of program status (PS).

![Figure 3.3-3 Program status (PS) Bit Configuration]

The reserved bits of program status (PS) are all reserved for future expansion. The read value of reserved bits is always "0". Write values should always be written as "0".
3.3.4 System Status Register (SSR)

System status register (SSR) is a 4-bit register that indicates the state of the CPU. It lies between bit 31 and bit 28 of the program status (PS).

Figure 3.3-4 shows the bit configuration of system status register (SSR).

**Figure 3.3-4 System Status Register (SSR) Bit Configuration**

<table>
<thead>
<tr>
<th>bit31</th>
<th>bit30</th>
<th>bit29</th>
<th>bit28</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBG</td>
<td>UM</td>
<td>FPU</td>
<td>MPU</td>
<td>0011B</td>
</tr>
</tbody>
</table>

The contents of each bit are described below.

- **[bit31] DBG: Debug State Flag**
  
  This flag indicates the debugging state during debugging. The flag bit is turned to "1" when the system shifts to a debug state, and turned to "0" when moving from the debug state with a RETI instruction. This cannot be rewritten using instructions such as the MOV instruction.

  The initial value of the debug state flag (DBG) after a reset is "0".

- **[bit30] UM: User Mode Flag**
  
  This flag indicates the user mode. The flag bit is turned to "1" when the system is shifted to user mode by the execution of a RETI instruction, and cleared to "0" when shifted to privilege mode with EIT. Upon execution of the RETI instruction, if bit 30 of the PS value is set to "1", a value returned from memory, the system shifts to user mode. This cannot be rewritten using instructions such as the MOV instruction.

  The initial value of the user mode flag (UM) after a reset is "0".

- **[bit29] FPU: FPU presence flag**
  
  This flag indicates that the floating point calculation unit (FPU) is installed. The flag bit is set to "1" if a FPU is installed, and "0" if it is not the case. This bit cannot be rewritten.

  **Table 3.3-1 FPU presence flag (FPU) in the system status register**

<table>
<thead>
<tr>
<th>flag</th>
<th>value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPU</td>
<td>0</td>
<td>With FPU (installed)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Without FPU (not installed)</td>
</tr>
</tbody>
</table>

- **[bit28] MPU: MPU presence flag**
  
  This flag indicates that the memory protection unit (MPU) is installed. The flag bit is set to "1" if a MPU is installed, and "0" if it is not the case. This bit cannot be rewritten.

  **Table 3.3-2 MPU presence flag (MPU) in the system status register**

<table>
<thead>
<tr>
<th>flag</th>
<th>value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPU</td>
<td>0</td>
<td>With MPU (installed)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Without MPU (not installed)</td>
</tr>
</tbody>
</table>
3.3.5 Interrupt Level Mask Register (ILM)

Interrupt level mask register (ILM) is a 5-bit register used to store the interrupt level mask value. It lies between bit20 to bit16 of the program status (PS).

Figure 3.3-5 shows the bit configuration of interrupt level mask register (ILM).

![Figure 3.3-5 Interrupt Level Mask Register (ILM) Bit Configuration](image)

The value stored in interrupt level mask register (ILM), is used in the level mask of an interrupt. When the interrupt enable flag (I) is "1", the value of interrupt level mask register (ILM) is compared to the level of the currently requested interrupt. If the value of interrupt level mask register (ILM) is greater (interrupt level is stronger), interrupt requested is accepted. Figure 3.3-6 shows the functions of interrupt level mask.

![Figure 3.3-6 Functions of Interrupt Level Mask](image)

The values of interrupt level range from 0(00000B) to 31(11111B). The smaller the value of interrupt level, the stronger it is, and the larger the value, the weaker it is. 0(00000B) is the strongest interrupt level, while 31(11111B) is the weakest.

There are following restrictions on values of the interrupt level mask register (ILM) that can be set from a program.

- When the value of interrupt level mask register (ILM) lies between 0(00000B) to 15(01111B), only values from 0(00000B) to 31(11111B) can be set.
- When the value of interrupt level mask register (ILM) lies between 16(10000B) to 31(11111B), only values between 16(10000B) to 31(11111B) can be set.
- When setting of values between 0(00000B) to 15(01111B) is attempted, 16 is added on automatically and values between 16(10000B) to 31(11111B) are set.

The interrupt level mask register (ILM) is initialized to 15(01111B) following a reset. If an interrupt request is accepted, the interrupt level corresponding to that interrupt is set in the interrupt level mask register (ILM).

For setting a value in interrupt level mask register (ILM) from a program, the STILM instruction is used.
3.3.6 Condition Code Register (CCR)

Condition code register (CCR) is an 8-bit register that indicates the status of instruction execution. It lies between bit7 to bit0 of the program status (PS).

Figure 3.3-7 shows the bit configuration of condition code register (CCR).

![Figure 3.3-7 Condition Code Register (CCR) Bit Configuration](image)

<table>
<thead>
<tr>
<th>bit7</th>
<th>bit6</th>
<th>bit5</th>
<th>bit4</th>
<th>bit3</th>
<th>bit2</th>
<th>bit1</th>
<th>bit0</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>Reserved</td>
<td>S</td>
<td>I</td>
<td>N</td>
<td>Z</td>
<td>V</td>
<td>C</td>
<td>--00XXXX B</td>
</tr>
</tbody>
</table>

The contents of each bit are described below.

[bit7, bit6] Reserved

These are reserved bits. Read value is always "0". Write value should always be "0".

[bit5] S: Stack Flag

This flag selects the stack pointer to be used as general-purpose register R15. When the value of stack flag (S) is "0", system stack pointer (SSP) is used, while when the value is "1", user stack pointer (USP) is used.

<table>
<thead>
<tr>
<th>flag</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td>System stack pointer (SSP)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>User stack pointer (USP)</td>
</tr>
</tbody>
</table>

If an EIT operation is accepted, stack flag (S) automatically becomes "0". However, the value of the condition code register (CCR) saved in system stack is the value which is later replaced by "0".

The initial value of stack flag (S) after a reset is "0".

[bit4] I: Interrupt Enable Flag

This flag is used to enable/disable mask-able interrupts. The value "0" of interrupt enable flag (I) disables an interrupt while "1" enables an interrupt. When an interrupt is enabled, the mask operation of interrupt request is performed by interrupt level mask register (ILM).

<table>
<thead>
<tr>
<th>flag</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>0</td>
<td>Interrupt disable</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Interrupt enable</td>
</tr>
</tbody>
</table>

The value of this flag is replaced by "0" by execution of INT instruction. However, the value of condition code register (CCR) saved in the system stack is the value which is later replaced by "0".

The initial value of an interrupt enable flag (I) after a reset is "0".
[bit3] N: Negative Flag

This flag is used to indicate positive or negative values when the results of a calculation are expressed in two’s complement form. The value "0" of the negative flag (N) indicates a positive value while "1" indicates a negative value.

<table>
<thead>
<tr>
<th>flag</th>
<th>value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>0</td>
<td>Calculation result is a positive value</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Calculation result is a negative value</td>
</tr>
</tbody>
</table>

The initial value of Negative flag (N) after a reset is undefined.

[bit2] Z: Zero Flag

This flag indicates whether the result of a calculation is zero or not. The value "0" of zero flag (Z) indicates a non-zero value, while "1" indicates a zero value.

<table>
<thead>
<tr>
<th>flag</th>
<th>value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>0</td>
<td>Calculation result is a non-zero value</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Calculation result is a zero value</td>
</tr>
</tbody>
</table>

The initial value of Zero flag (Z) after a reset is undefined.

[bit1] V: Overflow Flag

This flag indicates whether an overflow has occurred or not when the results of a calculation are expressed in two’s complement form. The value "0" of an overflow flag (V) indicates no overflow, while value "1" indicates an overflow.

<table>
<thead>
<tr>
<th>flag</th>
<th>value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>0</td>
<td>No overflow</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Overflow</td>
</tr>
</tbody>
</table>

Initial value of overflow flag (V) after a reset is indefinite

[bit0] C: Carry Flag

This flag indicates whether a carry or borrow condition has occurred in the highest bit of the results of a calculation. The value "0" of the carry flag (C) indicates no carry or borrow, while a value "1" indicates a carry or borrow condition.

<table>
<thead>
<tr>
<th>flag</th>
<th>value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
<td>No carry or borrow</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Carry or borrow condition</td>
</tr>
</tbody>
</table>

The initial value of a carry flag (C) after reset is undefined.
3.3.7 System Condition Code Register (SCR)

System condition code register (SCR) is a 3-bit register used to control the intermediate data of stepwise division and step trace trap. It lies between bit10 to bit8 of the program status (PS).

Figure 3.3-8 shows the bit configuration of system condition code register (SCR).

![Figure 3.3-8 System Condition Code Register (SCR) Bit Configuration](image)

The contents of each bit are described below.

[bit10, bit9] D1, D0: Step Intermediate Data

These bits are used for intermediate data in stepwise division. This register is used to assure resumption of division calculations when the stepwise division program is interrupted during processing.

If changes are made to the contents of the intermediate data (D1, D0) during division processing, the results of the division are not assured. If another processing is performed during stepwise division processing, division can be resumed by saving/retrieving the program status (PS) in/from the system stack.

Intermediate data (D1, D0) of stepwise division is made into a set by referencing the dividend and divisor by executing the "DIV0S" instruction. It is cleared by executing the "DIV0U" instruction.

The initial value of intermediate data (D1, D0) of stepwise division after a CPU reset is undefined.

[bit8] T: Step Trace Trap Flag

This flag specifies whether the step trace trap operation has to be enabled or not. When the step trace trap flag (T) is set to "1", step trace flag operation is enabled and the CPU generates an EIT event by trap operation after each instruction execution.

<table>
<thead>
<tr>
<th>flag</th>
<th>value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>0</td>
<td>Step trace trap disabled</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Step trace trap enabled</td>
</tr>
</tbody>
</table>

When the step trace trap flag (T) is "1", all NMI & user interrupts are disabled.

Step trace trap function uses an emulator. During a user program which uses the emulator, step trace trap function cannot be used (the emulator cannot be used for debugging in the step trace trap routine).

The initial value of step trace trap flag (T) after a reset is "0".
3.3.8 Return Pointer (RP)

Return pointer (RP) is a 32-bit register which stores the address for returning from a subroutine. It stores the program counter (PC) value upon execution of a CALL instruction.

Figure 3.3-9 shows the bit configuration of return pointer (RP).

**Figure 3.3-9 Return Pointer (RP) Bit Configuration**

<table>
<thead>
<tr>
<th>bit31</th>
<th>bit0</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>XXXXXXXXXX&lt;sub&gt;H&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

In case of a CALL instruction with a delay slot, the value stored in RP will be the address of the CALL instruction +4.

In case of a CALL instruction without a delay slot, the value stored in RP will be the address of the CALL instruction +4.

When returning from a subroutine by the RET instruction, the address stored in the return pointer (RP) is returned to the program counter (PC).

Return pointer (RP) does not have a stack configuration. When calling another subroutine from the subroutine called using the CALL instruction, it is necessary to first save the contents of the return pointer (RP) and restore them before executing the RET instruction.

Figure 3.3-10 shows a sample operation of the return pointer (RP) during the execution of a CALL instruction without a delay slot, and Figure 3.3-11 shows a sample operation of return pointer (RP) during the execution of a RET instruction.

**Figure 3.3-10 Sample Operation of RP during Execution of a CALL Instruction without a Delay Slot**
### 3.3.9 System Stack Pointer (SSP)

The system stack pointer (SSP) is a 32-bit register that indicates the address to be saved/restored to the system stack used at the time of EIT processing. The system stack pointer (SSP) is available when CPU is in privilege mode (UM=0).

Figure 3.3-12 shows the bit configuration of system stack pointer (SSP).

#### Figure 3.3-12 System Stack Pointer (SSP) Bit Configuration

<table>
<thead>
<tr>
<th>bit31</th>
<th>bit0</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>00000000H</td>
</tr>
</tbody>
</table>

When the stack flag (S) in the condition code register (CCR) is "0", the general-purpose register R15 is used as the system stack pointer (SSP). In a normal instruction, system stack pointer is used as the general-purpose register R15.

When an EIT event occurs, regardless of the value of the stack flag (S), the program counter (PC) and program status (PS) values are saved to the system stack area designated by system stack pointer (SSP). The value of stack flag (S) is stored in the system stack as program status (PS), and is restored from the system stack at the time of returning from the EIT event using RETI instruction.

System stack uses pre-decrement/post-decrement for storing and retrieving data. While saving data, after performing a data size decrement on the value of system stack pointer (SSP), it is written onto the address indicated by system stack pointer (SSP). While retrieving data, after the data is read from the address indicated by the system stack pointer (SSP), a data size increment is performed on the value of system stack pointer (SSP).

Figure 3.3-13 shows an example of system stack pointer (SSP) operation while executing instruction "ST R13,@-R15" when the stack flag (S) is set to "0".
### 3.3.10 User Stack Pointer (USP)

User stack pointer (USP) is a 32-bit register used to save/retrieve data to/from the user stack. The user stack pointer (USP) is available irrespective of the CPU: whether it is in privilege mode (UM=0) or in user mode (UM=1). In privilege mode, the stack pointer should be selected by rewriting the stack flag (S). In user mode, only the user stack pointer (USP) is available.

Figure 3.3-14 shows the bit configuration of user stack pointer (USP).

#### Figure 3.3-14 User Stack Pointer (USP) Bit Configuration

<table>
<thead>
<tr>
<th>bit31</th>
<th>bit0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial value XXXXXXXXH</td>
<td></td>
</tr>
</tbody>
</table>

When the stack flag (S) in the condition code register (CCR) is "1", the general-purpose register R15 is used as the user stack pointer (USP). In a normal instruction, user stack pointer (USP) is used as the general-purpose register R15.

User stack uses pre-decrement/post-decrement to save/retrieve data. While saving data, after performing a data size decrement on the value of user stack pointer (USP), it is written onto the address indicated by the user stack pointer (USP). While retrieving data, the data is read from the address indicated by the user stack pointer (USP), and a data size increment is performed on the value of user stack pointer (USP).

Figure 3.3-15 shows an example of user stack pointer (USP) operation while executing the instruction "ST R13, @-R15" when the stack flag (S) is set to "1".
3.3.11 Table Base Register (TBR)

Table base register (TBR) is a 32-bit register that designates the vector table containing the entry addresses for EIT operations.

Figure 3.3-16 shows the bit configuration of table base register (TBR).

![Figure 3.3-16 Table Base Register (TBR) Bit Configuration](image)

The address of the reference vector is determined by the sum of the contents of the table base register (TBR) and the vector offset corresponding to the EIT operation generated. Vector table layout is realized in word units. As the address of the calculated vector is in word units, the lower two bits of the resulting address value are explicitly read as “0”.

Figure 3.3-17 shows an example of table base register (TBR).

![Figure 3.3-17 Example of Table Base Register (TBR) Example](image)
The reset value of table base register (TBR) is 000F FC00\text{H}. Do not set a value above FFFF FC00\text{H} for the table base register (TBR).

Precautions:

If values greater than FFFF FC00\text{H} are assigned to the table base register (TBR), this operation may result in an overflow when summed with the offset value. An overflow in turn will result in vector access to the area 0000 0000\text{H} to 0000 03FF\text{H}, which can cause a program run away.

### 3.3.12 Multiplication/Division Register (MDH, MDL)

Multiplication/Division register (MDH, MDL) is a 64-bit register comprised of MDH represented by the higher 32 bits and MDL represented by the lower 32 bits. During multiplication, the product is stored. During division, the value set for the dividend and the quotient is stored.

Figure 3.3-18 shows the bit configuration of Multiplication/Division register (MDH, MDL).

**Figure 3.3-18 Multiplication/Division Register (MDH, MDL) Bit Configuration**

<table>
<thead>
<tr>
<th>bit31</th>
<th>bit0</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDH</td>
<td>XXXXXXXX\text{H}</td>
<td></td>
</tr>
<tr>
<td>MDL</td>
<td>XXXXXXXX\text{H}</td>
<td></td>
</tr>
</tbody>
</table>

The function of Multiplication/Division register (MDH, MDL) is different during a multiplication and during a division operation.

- **Function during Multiplication**
  
  In case of a 32 bit × 32 bit multiplication (MUL, MULU instruction), the calculation result of 64-bit length is stored in the product register (MDH, MDL) as follows.
  
  \[
  \text{MDH: higher 32 bits} \\
  \text{MDL: lower 32 bits}
  \]

  In case of a 16 bit × 16 bit multiplication (MULH, MULUH instruction), the calculation result of 32-bit length is stored in the product register (MDH, MDL) as follows.
  
  \[
  \text{MDH: undefined} \\
  \text{MDL: result 32 bits}
  \]

  Figure 3.3-19 shows an example of multiplication operation using Multiplication/Division register (MDH, MDL).
### Function during Division

Before starting the calculation, the dividend is stored in the Multiplication/Division register (MDH, MDL).

- **MDH**: don’t care
- **MDL**: dividend

When division is performed using any of the instructions DIV0S/DIV0U, DIV1, DIV2, DIV3, DIV4S meant for division, the result of division is stored in the Multiplication/Division register (MDH, MDL) as follows.

- **MDH**: remainder
- **MDL**: quotient

Figure 3.3-20 shows an example of division operation using Multiplication/Division register (MDH, MDL).
3.3.13 Base Pointer (BP)

The base pointer (BP) register is used for pointing in base pointer indirect addressing mode.

Figure 3.3-21 shows the bit configuration of base pointer (BP).

![Figure 3.3-21 Base Pointer (BP) Bit Configuration](image)

3.3.14 FPU Control Register (FCR)

FPU control register (FCR) is a 32-bit register used to control the FPU. It has a flag that indicates the settings and status of the FPU operation mode.

The FPU control register (FCR) consists of the following five parts:
- Floating point condition code (FCC)
- Rounding mode (RM)
- Floating point exception enable flag (EEF)
- Floating point exception accumulative flag (ECF)
- Floating point exception flag (CEF)

Figure 3.3-22 shows the bit configuration of FPU control register (FCR).

![Figure 3.3-22 FPU control register (FCR) Bit Configuration](image)

The reserved bits of the FPU control register (FCR) are all reserved for future expansion. The read value of reserved bits is always "0". The write value should always be "0".

Floating point condition code (FCC)

Floating point condition code (FCC) is a 4-bit register that stores the condition code of a floating point calculation result. It lies between bit 31 and bit 28 of the FPU control register (FCR).

Figure 3.3-23 shows the bit configuration of the floating point condition code (FCC).

![Figure 3.3-23 Floating point condition code (FCC) Bit Configuration](image)
The content of each bit are described below.

(bit31) E : E flag
This flag indicates that FRj and FRi are equal based on the floating point compare instruction (FCMP) results.

(bit30) L : L flag
This flag indicates that FRi is less than FRj based on the floating point compare instruction (FCMP) results.

(bit29) G : G flag
This flag indicates that FRi is greater than FRj based on the floating point compare instruction (FCMP) results.

(bit28) U : U flag
This flag indicates that no comparison can be made (Unordered) based on the floating point compare instruction (FCMP) results.

■ Rounding mode (RM)
Rounding mode (RM) is a 2-bit register that designates rounding mode of floating point calculation results. It lies between bit 19 and bit 18 of the FPU control register (FCR). In FR81 Family CPU, only rounding up to the nearest value (RM=00B) can be set.

Figure 3.3-24 shows the bit configuration of rounding mode (RM). Table 3.3-10 shows details of the rounding mode.

**Figure 3.3-24  Rounding mode (RM) Bit Configuration**

<table>
<thead>
<tr>
<th>bit19</th>
<th>bit18</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM1</td>
<td>RM0</td>
<td>XXB</td>
</tr>
</tbody>
</table>

**Table 3.3-10  Rounding mode**

<table>
<thead>
<tr>
<th>RM</th>
<th>Rounding mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00B</td>
<td>The nearest value</td>
</tr>
<tr>
<td>01B</td>
<td>0</td>
</tr>
<tr>
<td>10B</td>
<td>+∞</td>
</tr>
<tr>
<td>11B</td>
<td>-∞</td>
</tr>
</tbody>
</table>
Floating point exception enable flag (EEF)

Floating point exception enable flag (EEF) is a 6-bit register that enables exception occurrences of floating point calculation. It lies between bit 17 and bit 12 of the FPU control register (FCR).

Figure 3.3-25 shows the bit configuration of the floating point exception enable flag (EEF).

Figure 3.3-25 Floating point exception enable flag (EEF) Bit Configuration

<table>
<thead>
<tr>
<th>bit17</th>
<th>bit16</th>
<th>bit15</th>
<th>bit14</th>
<th>bit13</th>
<th>bit12</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>X</td>
<td>U</td>
<td>O</td>
<td>Z</td>
<td>V</td>
<td>XXXXXX B</td>
</tr>
</tbody>
</table>

The content of each bit are described below.

[b17] D : D flag
This is a unnormalized number input exception enable flag. When this bit has been set to "1", the FPU exception occurs upon input of an unnormalized number. When this bit has been set to "0", the unnormalized number is regarded as "0" for calculation purposes.

[b16] X : X flag
This is an inexact exception enable flag. When this bit is set to "1" and an inexact has occurred in the calculation result, FPU exception occurs. When this bit is set to "0", the value resulting from rounding up is written in the register.

[b15] U : U flag
This is an underflow exception enable flag. When this bit is set to "1" and an underflow has occurred in the calculation result, FPU exception occurs. When this bit is set to "0", a value "0" is written in the register.

[b14] O : O flag
This is an overflow exception enable flag. When this bit is set to "1" and an overflow has occurred in the calculation result, FPU exception occurs. When this bit is set to "0", ±∞ or ±MAX is written in the register in accordance with the rounding mode (RM).

[b13] Z : Z flag
This is a division-by-zero exception enable flag. When this bit is set to "1" and division-by-zero is carried out, FPU exception occurs. When this bit is set to "0", infinite (∞), which indicates that the calculation has been carried out appropriately, is written in the register.

[b12] V : V flag
This is an invalid calculation exception enable flag. When this bit is set to "1" and an invalid calculation is carried out, FPU exception occurs. When this bit is set to "0", QNaN is written in the register in the calculation type instruction, ± MAX is written in the register in the conversion instruction, and "1" (unordered) is set for the U flag of the floating point condition code (FCC) in the compare instruction.
Floating point exception accumulative flag (ECF)

Floating point exception accumulative flag (ECF) is a 6-bit register that indicates the accumulative number of occurrences of floating point calculation exceptions. It lies between bit 11 and bit 6 of the FPU control register (FCR). Only a "0" can be written in the accumulative flags. The flag value will not be changed when "1" is written in the accumulative flags. The write value is evaluated by bit.

Figure 3.3-26 shows the bit configuration of the floating point exception accumulative flag (ECF).

**Figure 3.3-26 Floating point exception accumulative flag (ECF) Bit Configuration**

<table>
<thead>
<tr>
<th>bit11</th>
<th>bit10</th>
<th>bit9</th>
<th>bit8</th>
<th>bit7</th>
<th>bit6</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>X</td>
<td>U</td>
<td>O</td>
<td>Z</td>
<td>V</td>
<td>XXXXXB</td>
</tr>
</tbody>
</table>

The content of each bit are described below.

- **[bit11] D : D flag**
  This flag indicates that an unnormalized number has been entered while the unnormalized number input exception is disabled (EEF:D=0). This is a accumulative flag.

- **[bit10] X : X flag**
  This flag indicates that the calculation result has become inexact while the inexact exception is disabled (EEF:X=0). This is a accumulative flag.

- **[bit9] U : U flag**
  This flag indicates that an underflow has occurred in the calculation result while the underflow exception is disabled (EEF:U=0). This is a accumulative flag.

- **[bit8] O : O flag**
  This flag indicates that an overflow has occurred in the calculation result while the overflow exception is disabled (EEF:O=0). This is a accumulative flag.

- **[bit7] Z : Z flag**
  This flag indicates that a division by zero has occurred while the division-by-zero exception is disabled (EEF:Z=0). This is a accumulative flag.

- **[bit6] V : V flag**
  This flag indicates that an invalid calculation has been carried out while the invalid calculation exception is disabled (EEF:V=0). This is a accumulative flag.
Floating point exception flag (CFE) is a 6-bit register that indicates the exception occurrence of floating point calculation. It lies between bit 5 and bit 0 of the FPU control register (FCR). Each flag is set according to the calculation result. Each flag shall be cleared using software. Each flag can be set only to "0", and writing "1" to the flag is invalid. The write value is evaluated by bit. If the flag has not been cleared during exception processing, each flag is cumulated.

Figure 3.3-27 shows the bit configuration of the floating point exception flag (CFE).

**Figure 3.3-27  Floating point exception flag (CFE) Bit Configuration**

<table>
<thead>
<tr>
<th>bit5</th>
<th>bit4</th>
<th>bit3</th>
<th>bit2</th>
<th>bit1</th>
<th>bit0</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>X</td>
<td>U</td>
<td>O</td>
<td>Z</td>
<td>V</td>
<td>XXXXXB</td>
</tr>
</tbody>
</table>

The content of each bit are described below.

- **[bit5] D : D flag**
  This flag is set when an unnormalized number has been input while the unnormalized number input exception is enabled (EEF:D=1).

- **[bit4] X : X flag**
  This flag is set when the calculation result has become inexact while the inexact exception is enabled (EEF:X=1).

- **[bit3] U : U flag**
  This flag is set when an underflow has occurred in the calculation result while the underflow exception is enabled (EEF:U=1).

- **[bit2] O : O flag**
  This flag is set when an overflow has occurred in the calculation result while the overflow exception is enabled (EEF:O=1).

- **[bit1] Z : Z flag**
  This flag is set when a division by zero has occurred while the division-by-zero exception is enabled (EEF:Z=1).

- **[bit0] V : V flag**
  This flag is set when an invalid calculation has been carried out while the invalid calculation exception is enabled (EEF:V=1).
3.3.15 Exception status register (ESR)

This is a 32-bit register that indicates the balance of process when an exception occurs while executing the invalid instruction exception source and the multiple load/store instruction.

The exception status register (ESR) consists of the following two parts:

- Register list (RL)
- Invalid instruction exception source (INV)

Figure 3.3-28 shows the bit configuration of the exception status register (ESR).

### Figure 3.3-28 Exception status register (ESR) Bit Configuration

<table>
<thead>
<tr>
<th>bit31</th>
<th>bit16</th>
<th>bit15</th>
<th>bit7</th>
<th>bit6</th>
<th>bit0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RL</td>
<td>Reserved</td>
<td>RL15</td>
<td>RL0</td>
<td>0000H</td>
<td></td>
</tr>
</tbody>
</table>

The reserved bits of the exception status register (ESR) are all reserved for future expansion. The read value of reserved bits is always "0". Write value should always be "0".

#### Register List (RL)

Register list (RL) is a 16-bit register that indicates registers whose transmission has not ended when an exception occurs while a LDM0, LDM1, STM0, STM1, FLDM, or FSTM instruction is executed. It lies between bit 31 and bit 16 of the exception status register (ESR). The register list (RL) value is updated only when an exception occurs while a LDM0, LDM1, STM0, STM1, FLDM, or FSTM instruction is executed.

Figure 3.3-29 shows the bit configuration of the register list (RL), and Table 3.3-11 shows the correspondence between the register list (RL) bits and the registers.

### Figure 3.3-29 Register List (RL) Bit Configuration

<table>
<thead>
<tr>
<th>bit31</th>
<th>bit16</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RL15</td>
<td>RL0</td>
<td>0000H</td>
</tr>
</tbody>
</table>

### Table 3.3-11 Correspondence between the register list (RL) bits and the registers

<table>
<thead>
<tr>
<th>bit of ESR register</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>RL bit</td>
<td>RL15</td>
<td>RL14</td>
<td>RL13</td>
<td>RL12</td>
<td>RL11</td>
<td>RL10</td>
<td>RL9</td>
<td>RL8</td>
</tr>
<tr>
<td>LDM1, LDM0 instruction</td>
<td>R15</td>
<td>R14</td>
<td>R13</td>
<td>R12</td>
<td>R11</td>
<td>R10</td>
<td>R9</td>
<td>R8</td>
</tr>
<tr>
<td>STM1, STM0 instruction</td>
<td>R0</td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R4</td>
<td>R5</td>
<td>R6</td>
<td>R7</td>
</tr>
<tr>
<td>FLDM instruction</td>
<td>FR15</td>
<td>FR14</td>
<td>FR13</td>
<td>FR12</td>
<td>FR11</td>
<td>FR10</td>
<td>FR9</td>
<td>FR8</td>
</tr>
<tr>
<td>FSTM instruction</td>
<td>FR0</td>
<td>FR1</td>
<td>FR2</td>
<td>FR3</td>
<td>FR4</td>
<td>FR5</td>
<td>FR6</td>
<td>FR7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ESR bit</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>RL bit</td>
<td>RL7</td>
<td>RL6</td>
<td>RL5</td>
<td>RL4</td>
<td>RL3</td>
<td>RL2</td>
<td>RL1</td>
<td>RL0</td>
</tr>
<tr>
<td>LDM1, LDM0 instruction</td>
<td>R7</td>
<td>R6</td>
<td>R5</td>
<td>R4</td>
<td>R3</td>
<td>R2</td>
<td>R1</td>
<td>R0</td>
</tr>
<tr>
<td>STM1, STM0 instruction</td>
<td>R8</td>
<td>R9</td>
<td>R10</td>
<td>R11</td>
<td>R12</td>
<td>R13</td>
<td>R14</td>
<td>R15</td>
</tr>
<tr>
<td>FLDM instruction</td>
<td>FR7</td>
<td>FR6</td>
<td>FR5</td>
<td>FR4</td>
<td>FR3</td>
<td>FR2</td>
<td>FR1</td>
<td>FR0</td>
</tr>
<tr>
<td>FSTM instruction</td>
<td>FR8</td>
<td>FR9</td>
<td>FR10</td>
<td>FR11</td>
<td>FR12</td>
<td>FR13</td>
<td>FR14</td>
<td>FR15</td>
</tr>
</tbody>
</table>
Invalid instruction exception source (INV)

Invalid instruction exception source (INV) is a 7-bit register that indicates the source causing an invalid instruction exception. It lies between bit 6 and bit 0 of the exception status register (ESR). Each flag is set only when the source occurs. Each flag shall be cleared using software. Each flag can be set only to "0", and writing "1" to the flag is invalid. The write value is evaluated by bit.

Figure 3.3-30 shows the bit configuration of the invalid instruction exception source (INV).

**Figure 3.3-30 Invalid instruction exception source (INV) Bit Configuration**

<table>
<thead>
<tr>
<th>bit6</th>
<th>bit0</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DT</td>
<td>FPU</td>
<td>0000000B</td>
</tr>
<tr>
<td>IF</td>
<td>PI</td>
<td></td>
</tr>
<tr>
<td>FPU</td>
<td>SPR</td>
<td></td>
</tr>
<tr>
<td>PI</td>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>PR</td>
<td>RI</td>
<td></td>
</tr>
</tbody>
</table>

The content of each bit are described below.

- **[bit6] DT**: Data access error
  This flag is set when a bus error occurs during data access to a buffer-disabled area, or a system register is accessed in user mode.

- **[bit5] IF**: Instruction fetch error
  This flag is set when a bus error occurs during instruction fetch, and the instruction is executed.

- **[bit4] FPU**: FPU absence error
  This flag is set when an floating point type instruction is executed on a model without FPU installed.

- **[bit3] PI**: Privilege instruction execution
  This flag is set when a RETI or STILM instruction is executed in user mode.

- **[bit2] SPR**: System-dedicated register access
  This flag is set when a MOV or LD instruction is executed to the table base register (TBR), system stack pointer (SSP), or the exception status register (ESR) in user mode.

- **[bit1] DS**: Invalid instruction placement on delay slot
  This flag is set when an instruction that cannot be placed on delay slot is executed on the delay slot.

- **[bit0] RI**: Undefined instruction
  This flag is set when an undefined instruction code is being executed.
3.3.16 Debug Register (DBR)

The debug register (DBR) is a dedicated register accessible only in the debug state. Writing to this register other than in debug state is regarded as invalid.

Figure 3.3-31 shows the bit configuration of Debug Register (DBR).

**Figure 3.3-31 Debug Register (DBR) Bit Configuration**

<table>
<thead>
<tr>
<th>bit31</th>
<th>bit0</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXXX</td>
<td>XXXX</td>
<td>H</td>
</tr>
</tbody>
</table>
3.4 Floating-point Register

Floating point registers are using that store results for floating point calculations.

The floating-point register is 16, each having 32-bit length. As for the register, the name of FR0 to FR15 is named.

Figure 3.4-1 shows the construction and the initial value of the floating-point register.

**Figure 3.4-1 The construction and the initial value of the floating-point register**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR0</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR1</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR2</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR3</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR4</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR5</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR6</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR7</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR8</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR9</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR10</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR11</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR12</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR13</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR14</td>
<td>XXXX XXXXH</td>
</tr>
<tr>
<td>FR15</td>
<td>XXXX XXXXH</td>
</tr>
</tbody>
</table>
CHAPTER 4
RESET AND "EIT"
PROCESSING

This chapter describes reset and EIT processing in the FR81 family CPU. EIT processing is the generic name for exceptions, interrupt and trap.

4.1 Reset
4.2 Basic Operations in EIT Processing
4.3 Processor Operation Status
4.4 Exception Processing
4.5 Interrupts
4.6 Traps
4.7 Multiple EIT processing and Priority Levels
4.8 Timing When Register Settings Are Reflected
4.9 Usage Sequence of General Interrupts
4.10 Precautions
4.1 Reset

A reset forcibly terminates the current process, initializes the device, and restarts the program from the reset vector entry address.
The reset process is executed in privilege mode. Transition to user mode should be carried out by executing a RETI instruction.

When a reset is generated, CPU terminates the processing of the instruction execution at that time and goes into inactive status until the reset is cancelled. When the reset is cancelled, the CPU initializes all internal registers and starts execution beginning with the program indicated by the new value of the program counter (PC).

Reset processing has a higher priority level than each operation of the EIT processing described later. Reset is accepted even in between an EIT processing.

When a reset is generated, FR81 family CPU makes an attempt to initialize each register, but all registers cannot be initialized. Each register sets a value through the program executed after a reset, and uses it. Table 4.1-1 shows the registers that are initialized following a reset.

Table 4.1-1 Registers that are initialized following a reset

<table>
<thead>
<tr>
<th>Register</th>
<th>Initial Value</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program counter (PC)</td>
<td>Word data at location 0000FFFFH</td>
<td>Reset vector</td>
</tr>
<tr>
<td>Interrupt level mask register (ILM)</td>
<td>15(01111B)</td>
<td></td>
</tr>
<tr>
<td>Step trace trap flag (T)</td>
<td>&quot;0&quot;</td>
<td>Trace OFF</td>
</tr>
<tr>
<td>Interrupt enable flag (I)</td>
<td>&quot;0&quot;</td>
<td>Interrupt disabled</td>
</tr>
<tr>
<td>Stack flag (S)</td>
<td>&quot;0&quot;</td>
<td>Use SSP</td>
</tr>
<tr>
<td>Table base register (TBR)</td>
<td>0000F000H</td>
<td></td>
</tr>
<tr>
<td>System stack pointer (SSP)</td>
<td>00000000H</td>
<td></td>
</tr>
<tr>
<td>Debug state flag (DBG)</td>
<td>&quot;0&quot;</td>
<td>No debug state</td>
</tr>
<tr>
<td>User mode flag (UM)</td>
<td>&quot;0&quot;</td>
<td>Privilege mode</td>
</tr>
<tr>
<td>Exception status register (ESR)</td>
<td>00000000H</td>
<td></td>
</tr>
<tr>
<td>General-purpose register R15</td>
<td>SSP</td>
<td>As per stack flag (S)</td>
</tr>
</tbody>
</table>

For details of My computer built-in functions (peripheral devices, etc.) following a reset, refer to the Hardware Manual provided with each device.
4.2 Basic Operations in EIT Processing

Exceptions, interrupts and traps are similar operations applied under partially different conditions. They save information for terminating or restarting the execution of instructions and perform branching to a processing program.

4.2.1 Types of EIT Processing and Prior Preparation

EIT processing is a method which terminates the currently executing process and transfers control to a predetermined processing program after saving restart information to the memory. EIT processing programs can return to the prior program by use of the RETI instruction.

EIT processing operates in essentially the same manner for exceptions, interrupts and traps, with a few minor differences listed below by which it differentiates them.

- Exceptions are related to the instruction sequence, and processing is designed to resume from the instruction in which the exception occurred.
- Interrupts originate independently of the instruction sequence. Processing is designed to resume from the instruction immediately following the acceptance of the interrupt.
- Traps are also related to the instruction sequence, and processing is designed to resume from the instruction immediately following the instruction in which the trap occurred.

While performing EIT processing, apply to the following prior settings in the program.

- Set the values in vector table (defining as data)
- Set the value of system stack pointer (SSP)
- Set the value of table base register (TBR) as the initial address in the vector table
- Set the value of interrupt level mask register (ILM) above 16(10000B)
- Set the interrupt enable flag (I) to "1"

The setting of interrupt level mask register (ILM) and interrupt enable flag (I), will be required at the time of using interrupts.

To support the emulator debugger debug function, a processing called "break" is carried out in the user state of debugging. The processing differs from usual EIT. The following shows the sources causing "break". The break processing is executed when the source is detected in the user state.

- Instruction break exception
- Break interrupt
- Step trace trap
- INTE instruction execution
4.2.2 EIT Processing Sequence

FR81 family CPU processes EIT events as follows.

1. The vector table indicated by the table base register (TBR) and the offset value of the vector number corresponding to the particular EIT are used to determine the entry address for the processing program for the EIT.

2. For restarting, the contents of the old program counter (PC) and the old program status (PS) are saved to the stack area designated by the system stack pointer (SSP).

3. "0" is saved in the stack flag (S). Also, the interrupt level Mask Register (ILM) and interrupt enable flag (I) are updated through EIT.

4. Entry address is saved in the program counter (PC).

5. After the processing flow is completed, just before the execution of the instruction in the entry address, the presence of new EIT sources is determined.

Figure 4.2-1 shows the operations in the EIT processing sequence.

Vector tables are located in the main memory, occupying an area of 1 Kbyte beginning with the address shown in the table base register (TBR). This area is used as a table of entry addresses for EIT processing.

For details on vector tables, refer to "2.1.2 Vector Table Area" and "3.3.6 Condition Code Register (CCR)."

Regardless of the value of stack flag (S), the program status (PS) and program counter (PC) is saved in the stack pointed to by the system stack pointer (SSP). After an EIT processing has commenced, the program counter (PC) is saved in the address pointed to by the system stack pointer (SSP), while the program status (PS) is saved at address 4 plus the address pointed to by the system stack pointer (SSP).

Figure 4.2-2 shows an example of saving program counter (PC) and program status (PS) during the occurrence of an EIT event.
4.2.3 Recovery from EIT Processing

RETI instruction is used for recovery from an EIT processing program. The RETI instruction retrieves the value of program counter (PC) and program status (PS) from the system stack, EIT and recovers from the EIT processing.

1. Retrieving program counter (PC) from the system stack
   (SSP) → PC       SSP+4 → SSP

2. Retrieving program status (PS) from the system stack
   (SSP) → PS       SSP+4 → SSP

To ensure the program execution results after recovery from the EIT processing program, it is required that all contents of the CPU registers before the commencement of EIT processing program have been saved at the time of recovery. The registers used in the EIT processing programs should be saved in the system stack and retrieved just before the RETI instruction.
4.3 Processor Operation Status

Processor operation is comprised of four states: Reset, normal operation, low-power consumption, and debugging.

- **Reset state**
  
  A state where the CPU is being reset. Two levels are provided for the reset state: Initialize level and reset level. When an initialize level reset is issued, all functions inside the MCU chip are initialized. When a reset level is issued, functions except debug control, and some parts of the clock and reset controls are initialized.

- **Normal operation state**
  
  A state where the sequential instructions and EIT processing are currently executed. Privilege mode (UM=0) and user mode (UM=1) are provided for the normal operation state. Some instructions and access destinations are disabled in user mode while they are enabled in privilege mode.

  After release of a reset state, the system enters privilege mode in the normal operation state, and is shifted to user mode by executing a RETI instruction. In the normal operation state, user mode is shifted to privilege mode by executing reset or EIT, and privilege mode is shifted to user mode by executing a RETI instruction.

- **Low-power consumption state**
  
  A state where the CPU stops operating to save power consumption. Transition to the lower power consumption state is carried out by controlling the stand-by in the clock control section. Three modes are provided for the low-power consumption state: Sleep, stop and clock. An interrupt shall be used to restore the system from the low-power consumption state.

- **Debugging state**
  
  A state where an in-circuit emulator (ICE) is connected, and debug related functions are enabled. The debugging state is separated into a user state and a debug state. In principle, a debugging state shall be shifted to the other state via a reset. However, a normal operation state can be forcibly shifted to a debugging state.

  As is the case with the normal operation state, privilege mode (UM=0) and user mode (UM=1) are provided for the user state. However, when a break is executed for debugging the state is shifted to the debug state. It is carried out in privilege mode under the debug state, and all registers and whole memory area can be accessed by disabling the memory protection and other functions. The debug state is shifted to the user state by executing a RETI instruction.

  Figure 4.3-1 shows transition between the processor operation states.
Figure 4.3-1  Transition between processor operation states

- Reset state
- User mode
- Normal operation
- Low-power consumption mode
- Executing a transition sequence
- Privilege mode
- RETI
- EIT
- ICE not connected
- DSU indication
- Debug State
- Break
- RETI
- Debugging
- Low-power consumption mode
- Executing a transition sequence
- User State
- User mode
- Low-power consumption state
4.4 Exception Processing

Exceptions originate when there is a problem in the instruction sequence. Exceptions are processed by first saving the necessary information to resume the currently executing instruction, and then starting the processing routine corresponding to the type of exception that has occurred.

Branching to the exception processing routine takes place before execution of the instruction that has caused the exception. The address of the instruction in which the exception occurs becomes the program counter (PC) value that is saved to the stack at the time of occurrence of the exception.

The following factors can cause occurrence of an exception:

- Invalid instruction exception
- Instruction access protection violation exception
- Data access protection violation exception
- FPU exception
- Instruction break
- Guarded access break

4.4.1 Invalid Instruction Exception

An invalid instruction exception occurs when an invalid instruction is being executed. The following sources can cause the invalid instruction exception.

- Executing an undefined instruction code.
- Executing on delay slot an instruction that cannot be placed on the delay slot.
- Writing to a system-dedicated register (TBR, SSP, or ESR) in user mode (with MOV or LD instruction).
- Executing a privilege instruction (RETI or STILM) in user mode.
- Executing a floating point instruction while FPU is absent.
- Occurrence of a bus error during instruction fetch.
- Occurrence of a bus error or violation of system register access during data access to a buffer-disabled area.

The following operations are performed if an invalid-instruction exception is accepted.

1. Transition to privilege mode is carried out, and the stack flag (S) is cleared.
   
   "0" → UM   "0" → S

2. Contents of program status (PS) are saved to the system stack.
   
   SSP - 4 → SSP   PS → (SSP)

3. Contents of the program counter (PC) of an exception source instruction are saved to the system stack.
   
   SSP - 4 → SSP   PC → (SSP)
4. The program counter (PC) value is updated by referring to the vector table.

\[(TBR + 3C4H) \rightarrow PC\]

5. A new EIT event is detected.

The address saved to the system stack as a program counter (PC) value represents the instruction itself that caused the undefined instruction exception. When a RETI instruction is executed, the contents of the system stack should be rewritten with the exception processing routine so that the execution will either resume from the address of the instruction next to the instruction that caused the exception.

### 4.4.2 Instruction Access Protection Violation Exception

An instruction access protection exception occurs when an instruction is executed in an area protected by the memory protection function.

During debugging, this exception can be treated as a break source according to an indication from the debugger. In this case, the instruction access protection violation exception does not occur.

Upon acceptance of the instruction access protection violation exception, the following operations take place.

1. Transition to privilege mode is carried out, and the stack flag (S) is cleared.

   \["0" \rightarrow UM \quad "0" \rightarrow S\]

2. The contents of the program status (PS) are saved to the system stack.

   \[SSP - 4 \rightarrow SSP \quad PS \rightarrow (SSP)\]

3. The contents of the program counter (PC) of an exception source instruction are saved to the system stack.

   \[SSP - 4 \rightarrow SSP \quad PC \rightarrow (SSP)\]

4. The program counter (PC) value is updated by referring to the vector table.

   \[(TBR + 3E4H) \rightarrow PC\]

5. A new EIT event is detected.

### 4.4.3 Data Access Protection Violation Exception

A data access protection violation exception occurs when an invalid data access is executed in an area protected by the memory protection function.

During debugging, this exception can be treated as a break source according to an indication from the debugger. In this case, the data access protection violation exception does not occur.

If this exception occurs during data access with a RETI instruction in the process of an EIT sequence, the CPU stops operating and is capable of accepting a reset and break interrupt.
If this exception occurs while executing LDM0, LDM1, STM0, STM1, FLDM, or FSTM instruction, contents of execution until the occurrence are reflected in registers and memory. Check the register list (ESR:RL) for how far the instruction is executed.

Upon acceptance of the instruction access protection violation exception, the following operations take place.

1. Transition to privilege mode is carried out, and the stack flag (S) is cleared.
   
   "0" → UM       "0" → S

2. The contents of the program status (PS) are saved to the system stack.
   
   SSP - 4 → SSP   PS → (SSP)

3. The contents of the program counter (PC) of an exception source instruction are saved to the system stack.
   
   SSP - 4 → SSP   PC → (SSP)

4. The program counter (PC) value is updated by referring to the vector table.
   
   (TBR + 3E0H) → PC

5. A new EIT event is detected.

4.4.4 FPU Exception

An FPU exception occurs when a floating point instruction is executed. The occurrence of the floating point exception can be restrained with the floating point control register (FCR).

To prevent a subsequent instruction from being completed before detection of the FPU exception, when the FPU exception is enabled, a pipeline hazard should be generated in order to stall the pipeline. Thus the subsequent instruction will not pass the floating point instruction.

The following describes sources causing the FPU exception. For details on conditions of the occurrence, see the description of each instruction.

- When an unnormalized number has been input while the unnormalized number input is enabled.
- When the calculation result has become inexact while the inexact exception is enabled.
- When an underflow has occurred in the calculation result while the underflow exception is enabled.
- When an overflow has occurred in the calculation result while the overflow exception is enabled.
- When a division-by-zero operation has occurred while the division-by-zero exception is enabled.
- When an invalid calculation has been executed while the invalid calculation exception is enabled.

Upon acceptance of the FPU exception, the following operations take place.

1. Transition to privilege mode is carried out, and the stack flag (S) is cleared.
   
   "0" → UM       "0" → S

2. The contents of the program status (PS) are saved to the system stack.
   
   SSP - 4 → SSP   PS → (SSP)
3. The contents of the program counter (PC) of an exception source instruction are saved to the system stack.

\[ \text{SSP} - 4 \rightarrow \text{SSP} \quad \text{PC} \rightarrow (\text{SSP}) \]

4. The program counter (PC) value is updated by referring to the vector table.

\[ (\text{TBR} + 3E8H) \rightarrow \text{PC} \]

5. A new EIT event is detected.

### 4.4.5 Instruction Break

An instruction break generates an exception or a break based on address instructions given by the debug support unit (DSU). Upon detection of the instruction break in the user state during debugging, a break processing is carried out. Upon detection of the instruction break during normal operation, an exception processing is carried out.

The following describes the brake processing being carried out when the instruction break is accepted in the user state.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, 4 is set to the interrupt level mask register (ILM), and then the mode is shifted to the debug state.

\[ "0" \rightarrow \text{UM} \quad "0" \rightarrow \text{S} \quad "4" \rightarrow \text{ILM} \]

2. The contents of the program status (PS) are saved to the PS save register (PSSR).

\[ \text{PS} \rightarrow \text{PSSR} \]

3. The contents of the program counter (PC) of an exception source instruction are saved to the PS save register (PCSR).

\[ \text{PC} \rightarrow \text{PCSR} \]

4. An instruction is fetched from the emulator debug instruction register (EIDR1), and the handler is executed.

The following describes the exception processing being carried out when the instruction break is accepted during normal operation.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, and 4 is set to the interrupt level mask register (ILM).

\[ "0" \rightarrow \text{UM} \quad "0" \rightarrow \text{S} \quad "4" \rightarrow \text{ILM} \]

2. The contents of the program status (PS) are saved to the system stack.

\[ \text{SSP} - 4 \rightarrow \text{SSP} \quad \text{PS} \rightarrow (\text{SSP}) \]

3. The contents of the program counter (PC) of an exception source instruction are saved to the system stack.

\[ \text{SSP} - 4 \rightarrow \text{SSP} \quad \text{PC} \rightarrow (\text{SSP}) \]

4. The program counter (PC) value is updated by referring to the vector table.

\[ (\text{TBR} + 3D4H) \rightarrow \text{PC} \]

5. A new EIT event is detected.
4.4.6 Guarded Access Break

Guarded access break is a function that carries out a break processing instead of generating an exception when an instruction access protection violation or a data access protection violation occurs during debugging.

Whether each access protection violation is treated as a break or an exception processing is determined by the debugger. The guarded access break does not occur during normal operation.

If the debugger determines to carry out the break processing when instruction break has been accepted in the user state, the following operations are carried out.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, 4 is set to the interrupt level mask register (ILM), and then the mode is shifted to the debug state.
   
   "0" → UM  "0" → S  "4" → ILM

2. The contents of the program status (PS) are saved to the PS save register (PSSR).
   
   PS → PSSR

3. The contents of the program counter (PC) of an exception source instruction are saved to the PS save register (PCSR).
   
   PC → PCSR

4. An instruction is fetched from the emulator debug instruction register (EIDR1), and the handler is executed.
4.5 Interrupts

Interrupts originate independently of the instruction sequence. They are processed by saving the necessary information to resume the currently executing instruction sequence, and then starting the processing routine corresponding to the type of the interrupt that has occurred.

Instruction loaded and executing in the CPU before the interrupt will be executed till completion. However any instruction loaded in the pipeline after the interrupt will be cancelled. Hence, after completion of the interrupt processing, processing will return to the instruction following the generation of the interrupt signal.

The following four factors cause the generation of interrupts.

- General interrupts
- Non-maskable interrupt (NMI)
- Break interrupt
- Data access error interrupt

In case an interrupt is generated during the execution of stepwise division instructions, intermediate data is saved to the program status (PS) to enable resumption of processing. Therefore, if the interrupt processing program overwrites the contents of the program status (PS) data in the stack, the processor will resume the normal instruction operations following resumption of processing. However the results of the division calculation will be incorrect.

4.5.1 General interrupts

General interrupts originate as requests from in-built peripheral functions. Here, the in-built interrupt controller present in devices and external interrupt control units have been described as one of the peripheral functions.

The interrupt requests from various in-built peripheral functions are accepted via interrupt controller. There are some interrupt requests which use external interrupt control unit, taking external terminals as interrupt input terminals. Figure 4.5-1 shows the acceptance procedure of general interrupts.
Each interrupt request is assigned an interrupt level by the interrupt controller, and it is possible to mask requests according to their level values. Also, it is possible to disable all interrupts by using the interrupt enable flag (I) in the condition code register (CCR).

When interrupt requests are generated by peripheral functions, they can be accepted under the following conditions.

- The level of interrupt level mask register (ILM) is higher (i.e. the numerical value is smaller) than the interrupt level set in the interrupt control register (ICR) corresponding to the vector number
- The interrupt enable flag (I) in the condition code register (CCR) is set to “1”

Interrupt control register (ICR) is a register of interrupt controller. Refer to the hardware manual of various models for details about the interrupt controller.

The following operations are performed after a general interrupt is accepted.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, and the accepted interrupt request level is set to the interrupt level mask register (ILM).
   
   
   "0" → UM  "0" → S  Interrupt level → ILM

2. The contents of the program status (PS) are saved to the system stack.

   SSP - 4 → SSP  PS → (SSP)

3. The address of the instruction next to that accepted a general interrupt is saved to the system stack.

   SSP - 4 → SSP  next instruction address → (SSP)

4. The program counter (PC) value is updated by referring to the vector table.

   (TBR + Offset) → PC

5. A new EIT event is detected.

When using general interrupts, it is required to set the interrupt level in the interrupt control register (ICR) corresponding to the vector number of the interrupt controller. Also perform the settings of the various peripheral functions and interrupt enable. Refer to the hardware manual of each model for details on interrupt controller and various peripheral functions.
4.5.2 Non-maskable Interrupts (NMI)

Non-maskable interrupts (NMI) are interrupts that cannot be masked.

Depending upon the product series, there are models which do not support NMI (there are no external NMI terminals). Refer to the hardware manual of various models to check whether NMI is supported or not.

Even if the acceptance of interrupts have been restricted by setting of "0" in the interrupt enable flag (I) of the condition code register (CCR), interrupts generated by NMI cannot be restricted. The masking of interrupt level by the interrupt level mask register (ILM) is valid. If a value above 16(10000B) is set in the interrupt level mask register (ILM) by a program, normally "NMI" cannot be masked by the interrupt level.

The value of interrupt level mask register (ILM) is initialized to 15(01111B) following a reset. Therefore, NMI cannot be masked until a value above 16(10000B) by a program following a reset.

When an NMI is accepted, the following operations are performed.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, and 15 is set to the interrupt level mask register (ILM).

   "0"  →  UM  
   "0"  →  S    
   "15" →  ILM

2. The contents of the program status (PS) are saved to the system stack

   SSP - 4 →  SSP  
   PS →  (SSP)

3. The address of the instruction next to that accepted NMI is saved to the system stack.

   SSP - 4 →  SSP  
   next instruction address →  (SSP)

4. The program counter (PC) value is updated by referring to the vector table.

   (TBR + 3C0H) →  PC

5. A new EIT event is detected.

4.5.3 Break Interrupt

A break interrupt is used for break request from the debugger. The break interrupt is reported by a level, and accepted when the level is higher than that of the interrupt level mask register (ILM). The request levels from 0 to 31 are available. The level cannot be masked by the interrupt enable flag (I).

The following describes conditions to accept the break interrupt. When the conditions are met, the CPU accepts the break interrupt.

- When a break interrupt request level is higher than that of the interrupt level mask register (ILM)
- When the CPU is operating in the user state during debugging

The following describes the brake processing being carried out when the break interrupt is accepted.

1. Transition to privilege mode is carried out, the stack flag (S) cleared, 4 is set to the interrupt level mask register (ILM), and then the mode is shifted to the debug state.

   "0"  →  UM  
   "0"  →  S    
   "4"  →  ILM

2. The code event is determined for the instruction next to that accepted the break interrupt.
3. The contents of the program status (PS) are saved to the PS save register (PSSR).
   \[ \text{PS} \rightarrow \text{PSSR} \]

4. The contents of the program counter (PC) of the instruction next to that accepted the break interrupt are saved to the PC save register (PCSR).
   \[ \text{PC} \rightarrow \text{PCSR} \]

5. An instruction is fetched from the emulator debug instruction register (EIDR1), and the handler is executed.

### 4.5.4 Data Access Error Interrupt

Data access error interrupts occur when a bus error occurs during data access to the buffer enabled specified area. Data access error interrupts can be enabled/disabled using the data access error interrupt enable bit (MPUCR:DEE). After a data access error interrupt occurs, a new data access error interrupt will not occur until the data access error bit (DESR:DAE) is cleared.

The data access error interrupt acceptance conditions are described below.

- The data access error interrupt enable bit (MPUCR:DEE) is enabled.
- A bus error occurs during data access to the buffer enabled specified area.

The following operations are carried out if a data access error interrupt is accepted.

1. Transition to privilege mode is carried out, and the stack flag (S) is cleared.
   \[ "0" \rightarrow \text{UM} \quad "0" \rightarrow \text{S} \]

2. The contents of the program status (PS) are saved to the system stack.
   \[ \text{SSP} - 4 \rightarrow \text{SSP} \quad \text{PS} \rightarrow (\text{SSP}) \]

3. The contents of the program counter (PC) of the instruction which accepted the interrupt are saved to the system stack.
   \[ \text{SSP} - 4 \rightarrow \text{SSP} \quad \text{PC} \rightarrow (\text{SSP}) \]

4. The program counter (PC) value is updated.
   \[ (\text{TBR} + 3\text{DC}_{\text{H}}) \rightarrow \text{PC} \]

5. A new EIT event is detected.
Traps are generated from within the instruction sequence. Traps are processed by first saving the necessary information to resume processing from the next instruction in the sequence, and then starting the processing routine corresponding to the type of the trap that has occurred.

Branching to the processing routine takes place after execution of the instruction that has caused the trap. The address of the instruction in which the trap occurs becomes the program counter (PC) value that is saved to the stack at the time of trap generation.

Following factors can lead to generation of traps.
- INT instruction
- INTE instruction
- Step trace traps

### 4.6.1 INT Instructions

The "INT #u8" instruction is used to create a trap through software. It generates a trap corresponding to the interrupt number designated in the operand.

When the INT instruction is executed, the following operations take place.
1. Transition to privilege mode is carried out, and the stack flag (S) is cleared.
   
   \[
   "0" \rightarrow \text{UM} \quad \quad "0" \rightarrow S
   \]

2. The contents of the program status (PS) are saved to the system stack.
   
   \[
   \text{SSP} - 4 \rightarrow \text{SSP} \quad \text{PS} \rightarrow (\text{SSP})
   \]

3. The address of the next instruction is saved to the system stack.
   
   \[
   \text{SSP} - 4 \rightarrow \text{SSP} \quad \text{next instruction address} \rightarrow (\text{SSP})
   \]

4. The program counter (PC) value is updated by referring to the vector table.
   
   \[
   (\text{TBR} + 3\text{FC}_{16} - 4 \times \text{u8}) \rightarrow \text{PC}
   \]

5. A new EIT event is detected.

The value of program counter (PC) saved to the system stack represents the address of the next instruction after the INT instruction.

### 4.6.2 INTE Instruction

The INTE instruction is used to create a software trap for debugging. A trap does not occur when the system is in the debug state during debugging, or if the step trace trap flag (SCR:T) of the program status (PS) is set. The operation of the INTE instruction varies between the user state during debugging and normal operation.
The following operations are carried out when an INTE instruction is executed during normal operation.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, and 4 is set to the interrupt level mask register (ILM).
   
   "0" \rightarrow \text{UM} \quad "0" \rightarrow S \quad "4" \rightarrow \text{ILM}

2. The contents of the program status (PS) are saved to the system stack.
   
   \text{SSP - 4} \rightarrow \text{SSP} \quad \text{PS} \rightarrow (\text{SSP})

3. The contents of the program counter (PC) of the subsequent instruction are saved to the system stack.
   
   \text{SSP - 4} \rightarrow \text{SSP} \quad \text{next instruction address} \rightarrow (\text{SSP})

4. The program counter (PC) value is updated by referring to the vector table.
   
   (\text{TBR + 3D8H}) \rightarrow \text{PC}

5. A new EIT event is detected.

The following operations are carried out when an INTE instruction is executed in the user state during debugging.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, 4 is set to the interrupt level mask register (ILM), and then the mode is shifted to the debug state.
   
   "0" \rightarrow \text{UM} \quad "0" \rightarrow S \quad "4" \rightarrow \text{ILM}

2. The contents of the program status (PS) are saved to the PS save register (PSSR).
   
   \text{PS} \rightarrow \text{PSSR}

3. The contents of the program counter (PC) of the subsequent instruction are saved to the PS save register (PCSR).
   
   \text{PC} \rightarrow \text{PCSR}

4. An instruction is fetched from the emulator debug instruction register (EIDR1), and the handler is executed.

The address saved in the system stack as program counter (PC) represents the address of the next instruction after the "INTE" instruction.

The INTE instruction should not be used within a trap processing routine of step trace trap.

### 4.6.3 Step Trace Traps

Step trace traps are traps used for debugging programs. Through this, a trap can be created after the execution of each instruction by setting the step trace trap flag (T) in the system condition code register (SCR). The operation of the step trace trap varies between the user state during debugging and normal operation.

A step trace trap is accepted when an instruction for which the step trace trap flag (T) is changed from "0" to "1" is executed. A step trace trap does not occur when an instruction for which the step trace trap flag (T) is changed from "1" to "0" is executed. However, for RETI instructions, a step trace trap does not occur when a RETI instruction for which the step trace trap flag (T) is changed from "0" to "1" is executed.
A step trace trap is generated when the following conditions are met.

- Step trace trap flag (T) in the system condition code register (SCR) is set to "1".
- The currently executing instruction is not a delayed branching instruction
- User state in which CPU is in normal operation or debugging

Step trace trap is not generated immediately after the execution of a delayed branching instruction. It is generated after the execution of instruction within the delay slots.

When the step trace trap flag (T) is enabled, non-maskable interrupts (NMI) and general interrupts are disabled.

The following operations are carried out if a step trace trap is accepted during normal operation.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, the step trace trap flag (T) is cleared, and 4 is set to the interrupt level mask register (ILM).
   
   \[ "0" \rightarrow \text{UM} \quad "0" \rightarrow \text{S} \quad "0" \rightarrow \text{T} \quad "4" \rightarrow \text{ILM} \]

2. The contents of the program status (PS) are saved to the system stack.
   
   \[ \text{SSP} - 4 \rightarrow \text{SSP} \quad \text{PS} \rightarrow (\text{SSP}) \]

3. The contents of program counter (PC) of the next instruction is saved to the system stack.
   
   \[ \text{SSP} - 4 \rightarrow \text{SSP} \quad \text{next instruction address} \rightarrow (\text{SSP}) \]

4. The program counter (PC) value is updated by referring to the vector table.
   
   \[ (\text{TBR} + 3\text{CC_H}) \rightarrow \text{PC} \]

The address saved as program counter (PC) in the system stack represents the address of the next instruction after the step trace trap.

The following operations are carried out for the brake process when a step trace trap is accepted in the user state during debugging.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, the step trace trap flag (T) is cleared, 4 is set to the interrupt level mask register (ILM), and then the mode is shifted to the debug state.

   \[ "0" \rightarrow \text{UM} \quad "0" \rightarrow \text{S} \quad "0" \rightarrow \text{T} \quad "4" \rightarrow \text{ILM} \]

2. The contents of the program status (PS) are saved to the PS save register (PSSR).
   
   \[ \text{PS} \rightarrow \text{PSSR} \]

3. The contents of the program counter (PC) of the subsequent instruction are saved to the PS save register (PCSR).
   
   \[ \text{PC} \rightarrow \text{PCSR} \]

4. An instruction is fetched from the emulator debug instruction register (EIDR1), and the handler is executed.

- **Restrictions**

The INTE instruction should not be used within the step trace trap handler. Use the OCD step trace function for the device installed with OCD-DSU. Do not use the step trace trap explained in this section, instead but always write "0" for step trace trap flag (T).
4.7 Multiple EIT processing and Priority Levels

When multiple EIT requests occur at the same time, priority levels are used to select one source and execute the corresponding EIT sequence.

4.7.1 Multiple EIT Processing

When multiple EIT requests occur at the same time, CPU selects one source and executes the corresponding EIT sequence and then once applies EIT request detection for other sources before executing the instruction of the entry address, and this operation gets repeated.

At the time of EIT request detection, when all acceptable EIT sources have been exhausted, the CPU executes the processing routine of the EIT request accepted in the end.

When the processing is returned from the processing routine of the last EIT request accepted using the RETI instruction, the processing routine of the last but one EIT request is executed. When the processing is returned from the processing routine of the first accepted EIT request using the RETI instruction, the control returns to the user program after having processed a series of EIT processes. Figure 4.7-1 shows an example of multiple EIT processing.

Figure 4.7-1 Example of Multiple EIT Processing

For example, if A, B, C are three EIT requests that have occurred simultaneously, and have been accepted in the order of B, C, A, the execution of the processing routine will be in the order A, C, B.
4.7.2 Priority Levels of EIT Requests

The sequence of accepting each request and executing the corresponding processing routines when multiple EIT requests occur simultaneously, is decided by two factors - by the priority levels of EIT requests, and, how other EIT requests are to be masked when one EIT request has been accepted.

At the time when an EIT request occurs, and at the time of completion of an EIT sequence, the detection of EIT requests being generated at that time is performed, and which EIT request will be accepted will be decided. At the time of completion of the EIT sequence, the detection of EIT requests is carried out under the condition where masking has been done for the EIT sources other than the EIT request accepted just a while back. Table 4.7-1 shows the priority levels of EIT requests and masking of other sources.

Table 4.7-1 Priority Levels of EIT Requests & Masking of Other Sources

<table>
<thead>
<tr>
<th>Priority level</th>
<th>EIT Source</th>
<th>Masking of other sources</th>
<th>ILM after updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reset</td>
<td>Other sources discarded</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>Instruction break</td>
<td>All factors given lower priority</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Guarded access break</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Invalid instruction exception</td>
<td>All factors given lower priority</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Instruction access protection</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>exception</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data access protection exception</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FPU exception</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>INT instruction</td>
<td>I flag = 0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>INTE instruction</td>
<td>All factors given lower priority</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>General interrupt</td>
<td>ILM= level of source accepted</td>
<td>ICR</td>
</tr>
<tr>
<td>7</td>
<td>NMI</td>
<td>ILM=15</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>Data access error interrupt</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>Break interrupt</td>
<td>All factors given lower priority</td>
<td>Request level</td>
</tr>
<tr>
<td>10</td>
<td>Step Trace Traps</td>
<td>All factors given lower priority</td>
<td>4</td>
</tr>
</tbody>
</table>

There are times when the value of interrupt level mask register (ILM) gets modified due to the EIT request accepted earlier and the other EIT sources occurring simultaneously get masked and cannot be accepted. In such a case, until the processing routine of EIT sources that have occurred simultaneously have been executed and the control has returned to the user program, the user interrupt is suspended and is re-detected at the time of resumption of the user program.
4.7.3 EIT Acceptance when Branching Instruction is Executed

No interrupts are accepted when a branching instruction is executed for delayed branching instruction. Also, when an exception occurs in the delay slot, branching is cancelled, and the program counter (PC) for branching instruction is saved. Interrupts and traps are accepted for delay slot instruction. Table 4.7-2 shows the EIT acceptance and saved PC value for branching instructions.

Table 4.7-2 EIT acceptance and saved PC value for branching instruction

<table>
<thead>
<tr>
<th>EIT acceptance instruction</th>
<th>Branching instruction</th>
<th>Delay slot instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIT type</td>
<td>Exception</td>
<td>Interrupt/trap</td>
</tr>
<tr>
<td>Branching</td>
<td>Acceptance</td>
<td>Saved PC value</td>
</tr>
<tr>
<td>Yes</td>
<td>None</td>
<td>PC</td>
</tr>
<tr>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>×</td>
<td>PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>None</td>
<td>PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>√</td>
<td>PC</td>
</tr>
</tbody>
</table>
The timing when the new values are reflected after the interrupt enable flag (I) of program status (PS) and the value of interrupt level mask register (ILM) are modified will be explained in this section.

4.8.1 Timing when the interrupt enable flag (I) is requested

The interrupt request (enable/disable) is reflected from the instruction which modifies the value of interrupt enable flag (I).

Figure 4.8-1 shows the timing of reflection of the interrupt enable flag (I) when interrupt enable is set to (I=1), and Figure 4.8-2 shows the timing of reflection of the interrupt enable flag (I) when interrupt disable is set to (I=0).

**Figure 4.8-1  Timing of reflection of interrupt enable flag (I) when interrupt enable is set to (I=1)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>I flag</th>
<th>Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORCCR #10H</td>
<td>1</td>
<td>Enable</td>
</tr>
<tr>
<td>Instruction</td>
<td>1</td>
<td>Enable</td>
</tr>
</tbody>
</table>

Interrupt enabled from here

**Figure 4.8-2  Timing of reflection of interrupt enable flag (I) when interrupt disable is set to (I=0)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>I flag</th>
<th>Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDCCR #EFH</td>
<td>0</td>
<td>Enable</td>
</tr>
<tr>
<td>Instruction</td>
<td>0</td>
<td>Disable</td>
</tr>
</tbody>
</table>

Interrupt disabled from here
4.8.2 Timing of Reflection of Interrupt Level Mask Register (ILM)

Acceptance to interrupt request is reflected from the instruction which modifies the value of interrupt level mask register (ILM).

Figure 4.8-3 shows the timing of reflection when the interrupt level mask register (ILM) is modified.

Figure 4.8-3 Timing of reflection when the Interrupt level mask register (ILM) is modified

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ILM</th>
<th>Interrupt reception</th>
</tr>
</thead>
<tbody>
<tr>
<td>STILM #set_ILM_B</td>
<td>B</td>
<td>B</td>
</tr>
</tbody>
</table>

"set_ILM_B" is a value of the interrupt level mask register (ILM) to be newly assigned. As in the case of STILM #30, assign a numeric value of 0 to 31.
4.9 Usage Sequence of General Interrupts

General interrupts accept interrupt requests from in-built peripheral functions and external terminals, and perform EIT processing. The general points of caution of programming while using general interrupts have been described here. Refer to the hardware manual of various models as the detailed procedure differs as per the peripheral function.

4.9.1 Preparation while using general interrupts

Before using general interrupts, settings for EIT processing need to be made. Perform the following settings in the program beforehand.

- Set values in the vector table (defined as data)
- Set up the system stack pointer (SSP) values
- Set up the table base register (TBR) value as the initial address in the vector table
- Set a value of above 16(10000B) in the interrupt level mask register (ILM)
- Set the value of "1" in the interrupt enable flag (I)

After the above settings, the settings of the peripheral functions are performed. In case of peripheral functions which use general interrupts, two bits in the register of the peripheral functions require to be set - a flag bit that indicates that a phenomenon which can become an interrupt source has occurred, and an interrupt enable bit which uses this flag bit to enable or disable the interrupt request.

The peripheral function verifies the operation halt status, the disable of interrupt request, and that the flag bit has been cleared. This state is achieved following a reset.

In case the peripheral function is engaged in some operation, the interrupt request is disabled and the flag bit cleared after the operation of the peripheral function has been halted.

The interrupt level is set in the interrupt control register (ICR) of the interrupt controller. As multiple interrupt control registers (ICR) are available corresponding to various vector numbers in the, please set an interrupt control register (ICR) corresponding to the vector number of the interrupt begin used.

The operation of the peripheral function is resumed after clearing the flag bit and enabling the interrupt request.
4.9.2 Processing during an Interrupt Processing Routine

After the interrupt request for a general interrupt has been accepted in the CPU as EIT following its generation, the control moves to the interrupt processing routine after the execution of the EIT sequence.

Vector numbers are assigned to each source of general interrupts, and the interrupt processing routine corresponding to these vector numbers are started. The interrupt sources and vector numbers do not necessarily have a one-to-one correspondence, and at times the same vector number is assigned to multiple interrupt sources. In such a case, the same interrupt processing routine is used for multiple interrupt sources.

Right in the beginning of the interrupt processing routine, the flag bit which indicates an interrupt source is verified. If the flag bit has been set, interrupt request for that interrupt is generated and the required processing (program) is executed after clearing the flag bit. In case, the same vector offset is being used for multiple interrupt sources, there are multiple flag bits indicating interrupt sources, and each of them are identified and processed in the same manner.

It is necessary to clear the flag bit while the interrupt of that particular interrupt source is in the disabled state. When the interrupt processing routine is started after the execution of the EIT sequence, the interrupt level of the general interrupt is stored in the interrupt level mask register (ILM) and the general interrupt of that interrupt level is disabled. Make sure to clear the flag bit at the end of the interrupt processing without modifying the interrupt level mask register (ILM).

The control is returned from the interrupt processing routine by the RETI instruction.

4.9.3 Points of Caution while using General Interrupts

Interrupt requests are enabled either when the corresponding flag bit has been cleared, or at the time of clearing the flag bit. Enabling interrupt requests when the flag bit is in the set state, leads to the generation of interrupt request immediately.

While enabling interrupt requests, do not clear flag bit besides the interrupt processing routine. Flag bit should be cleared at the time of disabling interrupt request.

In case a flag bit is cleared when a peripheral function is performing an operation, there are times when the flag bit cannot be cleared if the clearing of flag bit by writing to the register and the occurrence of a phenomenon which can be an interrupt source take place simultaneously or at a very close interval. Whether a flag bit will be cleared or not when the clearing of flag bit and the occurrence of a phenomenon that can become an interrupt source take place simultaneously, differs from one peripheral function to the other.
The precautions of The reset and the EIT processing described here.

4.10.1 Exceptions in EIT Sequence and RETI Sequence

If a data access protection violation exception (including guarded access break) or invalid instruction exception (data access error) occurs in the EIT or RETI sequence, since access to the system stack area is disabled, the CPU goes into the inactive state. To restore the system from this state, reset the system or execute a break interrupt from the debugger.

At this time, the data access protection violation exception or invalid instruction exception (data access error) cannot be accepted, and the processing is stopped immediately. The reset process/break process starts when a reset or break request is detected in the CPU stopped. If the CPU shifts to this stop state, since the EIT or RETI sequence is stopped in the middle of execution, it is impossible to execute the user program with this condition.

4.10.2 Exceptions in Multiple Load and Multiple Store Instructions

If a data access protection violation exception, invalid instruction exception (data access error), or guarded access break (data access) occurs when the LDM0, LDM1, STM0, STM1, FLDM or FSTM instruction is executed, the result of the processing up to this point is reflected in the memory or R15 (SSP/USP). The list of registers that have not been executed is stored in the register list of the exception status register (ESR).

4.10.3 Exceptions in Direct Address Transfer Instruction

If a data access protection violation exception, invalid instruction exception (data access error), or guarded access break (data access) occurs when data is transferred from the direct area to the memory by the direct address transfer instruction (DMOV), the I/O register value is updated when the I/O register value in the direct area is changed by reading.
CHAPTER 5

PIPELINE OPERATION

This chapter explains the chief characteristics of FR81 family CPU like pipeline operation, delayed branching processing etc.

5.1 Instruction execution based on Pipeline
5.2 Pipeline Operation and Interrupt Processing
5.3 Pipeline hazards
5.4 Non-block loading
5.5 Delayed branching processing
CHAPTER 5  PIPELINE OPERATION

5.1 Instruction execution based on Pipeline

FR81 Family CPU processes a instruction using a pipeline operation. This makes it possible to process to process nearly all instructions in one cycle. FR81 Family has two pipelines: an integer pipeline and floating point pipeline.

Pipeline operation divides each type of step that carries out interpretation and execution of instructions of CPU into stages, and simultaneously executes different stages of each instruction. Instruction execution that requires multiple cycles in other processing methods is apparently conducted in one cycle here.

Processing of both the integer pipeline and floating point pipeline are common up to the decoding stage, and independent processing is carried out for each pipeline from the execution and subsequent stages. The process sequence for each pipeline differs from the sequence of issuing instructions. However, the processing result that has been acquired by following the program sequence procedure is guaranteed.

5.1.1 Integer Pipeline

Integer pipeline is a 5-stage pipeline compatible with FR family. A 4-stage load buffer is provided for non-blocking loading.

The integer pipeline has the following 5-stage configuration.

- **IF Stage**: Fetch Instruction
  Instruction address is generated and instruction is fetched.
- **ID Stage**: Decode Instruction
  Fetched instruction is decoded. Register reading is also carried out.
- **EX Stage**: Execute Instruction
  Computation is executed.
- **MA Stage**: Memory Access
  Loading or access to storage is executed against the memory.
- **WB Stage**: Write Back to register
  Computation result (or loaded memory data) is written in the register.

Example of the integer pipeline operation (1) is shown in Figure 5.1-1 and Example of integer pipeline operation (2) is shown in Figure 5.1-2.
In principle, execution of instructions is carried out at one instruction per cycle. However, multiple cycles are necessary for the execution of instruction in case of load store instruction accompanied by memory wait, non-delayed branching instruction, and multiple cycle instruction. The speed of instruction execution is also reduced in cases where there is a delay in the supply of instructions, such as internal conflict of bus in the CPU, instruction execution through external bus interface etc.

Normally, instructions are executed sequentially in the integer pipeline. For example, if instruction A enters the pipeline before instruction B, it invariably reaches WB stage before instruction B. However, when the register used in Load instruction (LD instruction) is not used in the subsequent instruction, the subsequent instruction is executed before the completion of execution of load instruction based on the non-blocking loading buffer.

In the Example given in Figure 5.1-1, loading is carried out in R1 (load value is written in R1) based on preceding LD instruction, and R1 contents are referred to in the subsequent CMP instruction. Since the loaded data returns in 1 cycle, execution of instructions is sequential.

Similarly in the Example given in Figure 5.1-2, R1 that writes load value with LD instruction is used in the CMP instruction. Since the loaded data does not return in 1 cycle, execution till LDI:8 instruction is carried out and CMP instruction is made to wait at the ID stage by register hazard.
5.1.2 Floating Point Pipeline

The floating point pipeline is a 6-stage pipeline used to execute floating point calculations. The IF stage and ID stage are common with the integer pipeline.

The floating point pipeline has the following 5-stage configuration.

- **IF Stage: Fetch Instruction**
  Instruction address is generated and instruction is fetched.
- **ID Stage: Decode Instruction**
  Fetched instruction is decoded. Register reading is also carried out.
- **E1 Stage: Execute Instruction 1**
  Computation is executed. Multiple cycles may be required depending on the instruction.
- **E1 Stage: Execute Instruction 2**
  The result is rounded and normalized.
- **WB Stage: Write Back to register**
  Computation result is written in the register.
It is possible at times that an event wherein it appears that interrupt request is lost after acceptance of interrupt, if the flag that causes interrupt in the interrupt-enabled condition, because pipeline operation is conducted, occurs.

5.2.1 Mismatch in Acceptance and Cancellation of Interrupt

Because CPU is carrying out pipeline processing, pipeline processing of multiple instructions is already executed at the time of acceptance of interrupt. Therefore, in case corresponding interrupt cancellation processing among the instructions under execution in the pipeline (For example, clearing of flag bits that cause interrupt) is carried out, branching to corresponding interrupt processing program is carried out normally but when control is transferred to interrupt processing, the interrupt request is at times already over (Flag bits that cause interrupt having been cleared).

An Example of Mismatch in Acceptance and cancellation of interrupt is shown is Figure 5.2-1.

![Figure 5.2-1 Example of Mismatch in Acceptance and Cancellation of interrupt](image)

This type of phenomenon does not occur in case of exceptions and trap, because the operation for request cancellation cannot be carried out in the program.

5.2.2 Method of preventing the mismatched pipeline conditions

Mismatch in Acceptance and Deletion of interrupt can occur in case flag bits that cause interrupt are cleared while interrupt request is enabled in the peripheral functions.

To avoid such a phenomenon, programmers should set the interrupt enable flag (I) at "0", disable interrupt acceptance in CPU and clear the flag bits that cause an interrupt.
CHAPTER 5 PIPELINE OPERATION

5.3 Pipeline hazards

The FR81 Family CPU executes program steps in the order in which they are written and is therefore equipped with a function that detects the occurrence of data hazards and construction hazards, and stops pipeline processing when necessary.

5.3.1 Occurrence of data hazard

A data hazard occurs if dependency that refers or updates the register exists in between the preceding and subsequent instructions. The CPU may simultaneously process one instruction that involves writing values to a register, and a subsequent instruction that attempts to refer to the same register before the write process is completed.

An example of a data hazard is shown in Figure 5.3-1. In this case, the reading of R1 used as the address will read the value before the modification, as the read timing precedes the writing to R1 requested by the just previous instruction. (Actually, the data hazard is avoided, and the modified value is read.)

![Figure 5.3-1 Example of a data hazard](image1)

5.3.2 Register Bypassing

Even when a data hazard does occur, it is possible to process instructions without operating delays if the data intended for the register to be accessed can be extricated from the preceding instruction. This type of data transfer processing is called register bypassing.

An example of Register Bypassing is indicated in Figure 5.3-2. In this example, instead of reading the R1 in the ID stage of SUB instruction, the program uses the results of the calculation from ADD instruction (before the results are written to the register) and thus executes the instruction without delay.

![Figure 5.3-2 Example of a register bypass](image2)
5.3.3 Interlocking

Instructions that are relatively slow in loading data to the CPU may cause data hazards that cannot be handled by register bypassing.

In the example Figure 5.3-3, data required for the ID stage of the SUB instruction must be loaded to the CPU in the MA stage of the LD instruction, creating a data hazard that cannot be avoided by the bypass function.

Figure 5.3-3 Example: Data Hazard that cannot be avoided by Bypassing

```
LD @R0, R1  IF ID EX MA WB
SUB R1, R2  IF ID EX MA WB
```

In cases such as this, the CPU executes the instruction correctly by pausing before the execution of subsequent instruction. This function is called interlocking.

In the example in Figure 5.3-4, the ID stage of the SUB instruction is delayed until the data is loaded from the MA stage of the LD instruction.

Figure 5.3-4 Example of Interlocking

```
LD @R0, R1  IF ID EX MA WB
SUB R1, R2  IF ID EX MA WB
```

5.3.4 Interlocking produced by reference to R15 after Changing the Stack flag (S)

The general purpose register R15 is designed to function as either the system stack pointer (SSP) or user stack pointer (USP). For this reason the FR Family CPU is designed to automatically generate an interlock whenever a change to the stack flag (S) in the program status (PS) is followed immediately by an instruction that references the R15. This interlock enables the CPU to reference the SSP or USP values in the order in which they are written in the program.

Hardware design similarly generates an interlock whenever a TYPE-A format instruction immediately follows an instruction that changes the value of Stack flag (S). For information on instruction formats, see Section "6.2.3 Instruction Formats".

5.3.5 Structural Hazard

A structural hazard occurs if a resource conflict occurs between instructions which use the same hardware resource. If this hazard is detected, the pipeline is interlocked to pause the processing of subsequent instruction until the hazard is eliminated.
5.3.6 Control Hazard

A control hazard occurs when the next instruction cannot be fetched before the branching instruction is complete. In FR81 Family, to reduce penalty due to this control hazard, the pre-fetch function that bypasses the branch destination address from the ID stage and the delayed branching instruction have been added. Therefore, penalties do not become apparent.

● Pre-fetch function

FR81 Family CPU has a 32-bit 4-stage pre-fetch buffer, and fetches a subsequent instruction of consecutive addresses as long as the buffer is not full. However, when a branching instruction is decoded, the instruction is fetched from the branching destination regardless of the condition. If an instruction is branched, the instruction in the pre-fetch buffer is discarded, and the subsequent instruction in the branching destination will be pre-fetched. If not branched, the instruction in the branching destination is discarded, and the instruction in the pre-fetch buffer will be used.

● Delayed branching processing

Delayed branching processing is the function to execute the instruction immediately following the branching instruction for pipeline operation by one, regardless of whether the branching is successful or unsuccessful. The position immediately following a branching instruction is called the delay slot. Instructions that can be placed in the delay slot should be executable in one state having 16-bit length. Placing an instruction that does not fit in the delay slot will result an invalid instruction exception to occur. Refer to Appendix A.3 for the list of instructions that can be placed in delay slot.
Non-block loading is carried out in FR81 Family CPU. A maximum of 4 loading instructions can be issued with precedence.

In non-block loading, the subsequent instruction is executed without waiting for the completion of loading instruction, if the general-purpose register in which the load instruction value is stored is not referred in the subsequent instruction.

As shown below, when register R1 that stores data value based on LD instruction is referred to in the subsequent ADD instruction, the ADD instruction is executed after storing R1 value based on LD instruction.

```
LD @10,R1
ADD R1,R2 ; waits for completion of execution of preceding LD instruction
```

As shown below, ADD instruction is executed without waiting for the completion of execution of LD instruction when R1 that stores data value by LD instruction is not referred to in the subsequent ADD instruction. After that, at the time of execution of SUB instruction that references R1, if the preceding LD instruction is not already executed, the SUB instruction is executed after waiting for the completion of execution of that LD instruction.

```
LD @10,R1
ADD R2,R3 ; Does not wait for completion of execution of preceding LD instruction
SUB R1,R3 ; waits for completion of execution of preceding LD instruction
```

A maximum of 4 load instructions can be executed with precedence. It can also be used in the following way for issuing multiple LD instructions with precedence.

```
LD @100,R1 ; LD instruction (1)
LD @104,R2
LD @108,R3
LD @112,R4 ; a maximum of four LD instructions can be issued with precedence
ADD R5,R6 ; executed without waiting for the completion of execution of preceding LD instruction
SUB R6,R0
ADD R1,R5 ; executed after completion of execution of preceding LD instruction (1)
```
5.5 Delayed branching processing

Because FR81 Family CPU features pipeline operation, the loading of the instruction is already completed at the time of execution of branching instruction. The processing speeds can be improved by using the delayed branching processing.

5.5.1 Example of branching with non-delayed branching instructions

Non-delayed branching instruction executes instructions in the order of program but the execution speed drops down by 1 cycle as compared to delayed branching instruction when branching.

In a pipeline operation, by the time the CPU recognizes an instruction as a branching instruction the next instruction has already been loaded. To process the program as written, the instruction following the branching instruction must be cancelled in the middle of execution. Branching instructions that are handled in this manner are non-delayed branching instructions.

The example of processing non-delayed branching instruction with fulfilled branching conditions is given in Figure 5.5-1 which shows that execution of the "ST R2, @R12" instruction (instruction placed immediately after branching instruction) that had started pipeline operation before fetching instruction from the branching destination is cancelled in the middle. Due to this, program processing happens as the program is written, but branching instruction apparently takes 2 cycles for completion.

Figure 5.5-1 Example of processing of Non-Delayed Branching instruction (Branching conditions satisfied)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD @R10, R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD @R11, R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R1, R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE testOK (branching conditions satisfied)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R2, @R12 (instruction immediately after)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R2, @R13 (branch destination instruction)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- -- : Cancelled stages
- : PC change

Figure 5.5-2 shows an example of processing a non-delayed branching instruction when branching conditions are not fulfilled. In this example, the "ST R2, @R12" instruction (instruction kept immediately after branching instruction) that started pipeline processing before fetching instruction from the branching destination is executed without being cancelled. The processing of program happens as written in the program since the instructions are executed sequentially, without branching, and branching instruction execution speed is apparently of 1 cycle.
**5.5.2 Example of processing of delayed branching instruction**

Delayed branching instructions are processed with an apparent execution speed of 1 cycle, regardless of whether or not branching conditions are satisfied. When branching occurs, this is one cycle faster than using non-delayed branching instructions. However, the apparent order of instruction processing is inverted in cases where branching occurs.

An instruction immediately following a branching instruction will already be loaded by the CPU by the time the branching instruction is executed. This position is called the delay slot. A delayed branching instruction is a branching instruction that executes the instruction in the delay slot regardless of whether or not branching conditions are satisfied.

Figure 5.5-3 shows an example of processing a delayed branching instruction when branching conditions are satisfied. In this example, the branch destination instruction “ST R2,@R13” is executed after the instruction “ST R2,@R12” in the delay slot. As a result, the branching instruction has an apparent execution speed of 1 cycle. However, the instruction “ST R2,@R12” in the delay slot is executed before the branch destination instruction “ST R2,@R13” and therefore the apparent order of processing is inverted.

![Figure 5.5-2 Example of processing of Non-Delayed Branching instruction (Branching conditions not satisfied)](image)

**Figure 5.5-3 Example of processing of Delayed Branching instruction (Branching conditions satisfied)**
Figure 5.5-4 shows an example of processing a delayed branching instruction when branching conditions are not satisfied. In this example, the instruction “ST R2,@R12” in delay slot is executed without being cancelled. As a result, the program is processed in the order in which it is written. The branching instruction requires an apparent processing time of 1 cycle.

**Figure 5.5-4 Example of processing of Delayed Branching instruction**  
(Branching conditions not satisfied)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD @R10, R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD @R11, R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R1, R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE:D TestOK (branching conditions not satisfied)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R2, @R12 (delay slot instruction)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD #4, R12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Not canceled
CHAPTER 6

INSTRUCTION OVERVIEW

This chapter presents an overview of the instructions used with the FR81 Family CPU.

6.1 Instruction System
6.2 Instructions Formats
6.3 Data Format
6.4 Read-Modify-Write type Instructions
6.5 Branching Instructions and Delay Slot
6.6 Step Division Instructions
6.1 Instruction System

FR81 Family CPU has the integer type instruction of upward compatibility with FR80 Family and floating point type instruction executed by FPU.

6.1.1 Integer Type Instructions

Integer type instructions, in addition to instruction type of general RISC CPU, is also compatible with logical operation optimized for embedded use, bit operation and direct addressing instructions.

Integer type instructions of FR81 Family CPU can be divided into the following 15 groups.

- **Add/Subtract Instructions**
  These are the Instructions to carry out addition and subtraction between general-purpose registers or a general-purpose register and immediate data. They also enable computation with carry used in multi-word long computation or computations where flag value of Condition Code Register (CCR) convenient for address calculation is not changed.

- **Compare Instructions**
  These are the Instructions to carry out subtraction between general-purpose registers or a general-purpose register and immediate data and reflect the results in the flag of Condition Code Register (CCR).

- **Logical Calculation Instructions**
  These are the Instructions to carry out logical calculation for each bit between general-purpose registers or a general-purpose register and memory (including I/O). Logical calculation types are logical product (AND), logical sum (OR), and exclusive logical sum (EXOR). Memory addressing is register indirect.

- **Bit Operation Instructions**
  These are the Instructions to carry out logical calculation between memory (including I/O) and immediate value and operate directly for each bit. Logical calculation types are logical product (AND), logical sum (OR), and exclusive logical sum (EXOR). Memory addressing is register indirect.

- **Multiply/Divide Instructions**
  These are the instructions to carry out multiplication and division between general-purpose register and multiplication/division result register. There are 32 bit × 32 bit, 16 bit × 16 bit multiplication instructions and step division instructions to carry out 32 bit ÷ 32 bit division.

- **Shift Instructions**
  These are the instructions to carry out shift (logical shift, arithmetic shift) of general-purpose registers. By specifying general-purpose register or immediate data, shift (Barrel Shift) of multiple bits can be specified at once.
Immediate Data Transfer Instructions

These are the instructions to transfer immediate data to general-purpose registers and can transfer immediate data of 8bit, 20 bit, and 32 bit.

Memory Load Instructions

These are the instructions to load from memory (including I/O) to general-purpose registers or dedicated registers. They can transfer data length of 3 types namely, bytes, half-words and words and memory addressing is register indirect.

During memory addressing of some Instructions, Displacement Register Indirect or Increment/Decrement Register Indirect Address is possible.

Memory Store Instructions

These are the instructions to store from general-purpose register or dedicated register to memory (Including I/O). They can transfer data length of 3 types namely, bytes, half-words and words and memory addressing is register indirect.

During memory addressing of some Instructions, Displacement Register Indirect or Increment/Decrement Register Indirect Address is possible.

Inter-register Transfer Instructions/Dedicated Register Transfer Instructions

These are the instructions to transfer data between general-purpose registers or a general-purpose register and dedicated register.

Non-delayed Branching Instructions

These are the instructions that do not have delay slot and carry out branching, sub-routine call, interrupt and return.

Delayed Branching Instructions

These are the instructions that have delay slot and carry out branching, sub-routine call, interrupt and return. Delay slot instructions are executed when branching.

Direct Addressing Instructions

These are the instructions to transfer data between general-purpose register and memory (INCLUDING I/O) or between two memories. Addressing is not register indirect but direct specification with operand of instruction.

In some instructions, in combination with specific general-purpose registers, access is made in combination with increment/decrement Register Indirect addressing.
6.1 Bit Search Instructions

These are the instructions that have been added to FR81/FR80 Family CPU. They search 32-bit data of general-purpose register from MSB and obtain the first “1” bit, “0” bit and bit position of change point (distance of bit from MSB).

They correspond to bit search module packaged in the family prior to FR81/FR80 Family (FR30 Family, FR60 Family etc.) as peripheral function.

6.1.2 Floating Point Type Instructions

The floating point type instructions is an instruction added in FR81 family CPU. The floating point type instructions is divided into the following six groups.

- FPU Memory Load Instruction
  This is an instruction to load data from memory to the floating point register. Memory addressing is register indirect. Displacement Register Indirect or Increment/Decrement Register Indirect Address is possible.

- FPU Memory Store Instruction
  This is an instruction to store data from the floating point register in memory. Memory addressing is register indirect. Displacement Register Indirect or Increment/Decrement Register Indirect Address is possible.

- FPU Single-Precision Floating Point Calculation Instruction
  This is an instruction to perform single-precision floating point calculations.

- FPU Inter-Register Transfer Instruction
  This is an instruction to transfer data between floating point registers or between the floating point register and general-purpose register.

- FPU Branching Instruction without Delay
  This is a conditional branching instruction without a delay slot.

- FPU Branching Instruction with Delay
  This is a conditional branching instruction with a delay slot.
This part describes about Instruction Formats of FR81 Family CPU.

### 6.2.1 Instructions Notation Formats

#### ● Integer type instruction

The integer type instruction is 2 operand format. There are 3 types of Instruction notation formats depending on the number of operands. Instruction notation formats are as follows.

- **<Mnemonic> <Operand 1> <Operand 2>**
  
  Mnemonic calculations are carried out between operand 2 and operand 1 and the results are stored at operand 2.
  
  Ex: ADD R1, R2 ; R2 + R1 -> R2

- **<Mnemonic> <Operand 1>**
  
  Operations are designated by a mnemonic and use operand 1.
  
  Ex: JMP @R1 ; R1 -> PC

- **<Mnemonic>**
  
  Operations are designated by a mnemonic.
  
  Ex: NOP ; No Operation

Operands have general-purpose register, dedicated register, immediate data and combinations of part of general-purpose register and immediate data. Operand format varies depending on Instruction.

#### ● Floating point type instruction

Floating point type instruction is 3 operand format. The following description formats are added.

- **<Mnemonic> <Operand 1> <Operand 2> <Operand 3>**
  
  Mnemonic calculations are executed between operand 1 and operand 2 and the results are stored in operand 3. For some of the instructions, calculations are executed between operand 3 and the calculation result of operand 1 and operand 2, and then the results are stored in operand 3.
  
  Ex: FADDs FR1, FR2, FR3 ; FR1 + FR2 -> FR3
6.2.2 Addressing Formats

There are several methods for address specification when accessing memory in the memory space or I/O register. Addressing format varies depending on Instruction.

@General-purpose Registers

It is Register Indirect Addressing. Address indicated by the content of the general-purpose register is accessed.

@(R13, General-purpose Register)

Address where virtual accumulator (R13) and contents of general-purpose register are added is accessed.

@(R14, Immediate Data)

Address where contents of Frame Pointer (R14) and immediate data are added is accessed. Immediate data is specified in the multiples of data size (word, half word, byte).

@(R15, Immediate Data)

Address where contents of Stack Pointer (R15) and immediate data are added is accessed. Immediate data is specified in the multiples of data size (word, half word, byte).

@R15+

Write access to the address indicated by the contents of Stack Pointer (R15) is made. 4 will be added to the stack pointer (R15).

@-R15

Read access to the address which is deduction of 4 from the contents of Stack Pointer (R15) is made. 4 will be deducted from the Stack Pointer (R15).

@ Immediate Data

It is direct addressing. Address indicated by immediate data is accessed.

@R13+

Access to address indicated by the contents of virtual accumulator (R13) is made. Data size (Bytes) will get added to virtual accumulator (R13).

@(BP, Immediate Data)

Address where the base pointer (BP) and immediate data are added is accessed. Immediate data is specified in the multiples of data size (word, half word, byte).
6.2.3 Instruction Formats

FR81 Family CPU Instructions are 16-bit in length. Bit configuration of Instructions varies depending on configuration of operands of Instructions. Bit configuration of Instructions classified into groups is called Instruction Formats.

There are 14 types of Instructions Formats TYPE-A through TYPE-N.

TYPE-A
It has 8-bit OP Code (OP) and two Register designated fields (Rj/Rs, Ri)

<table>
<thead>
<tr>
<th>MSB</th>
<th>OP</th>
<th>Rj/Rs</th>
<th>Ri</th>
</tr>
</thead>
</table>

TYPE-B
It has 4-bit OP Code (OP) and 8-bit immediate data fields (i8/o8), register designated field (Ri)

<table>
<thead>
<tr>
<th>MSB</th>
<th>OP</th>
<th>i8/o8</th>
<th>Ri</th>
</tr>
</thead>
</table>

TYPE-C
It has 8-bit OP Code (OP) and 4-bit immediate data fields (u4/m4/i4), register designated field (Ri)

<table>
<thead>
<tr>
<th>MSB</th>
<th>OP</th>
<th>u4/m4/i4</th>
<th>Ri</th>
</tr>
</thead>
</table>

TYPE-D
It has 8-bit OP Code (OP) and 8-bit immediate data field (u8) or address designated field (rel8/dir8). In some instructions, it is 8-bit register list designated field (rlist).

<table>
<thead>
<tr>
<th>MSB</th>
<th>OP</th>
<th>u8/rel8/dir8/reglist</th>
<th>Ri</th>
</tr>
</thead>
</table>
CHAPTER 6 INSTRUCTION OVERVIEW

6.2

TYPE-E
It has 12-bit OP Codes (OP) and register designated fields (Ri/Rs).

<table>
<thead>
<tr>
<th>MSB</th>
<th>OP</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TYPE-E'
It is deformation of TYPE-E. It has 12-bit OP Code (OP). TYPE-E register designated field is fixed to 0000B. It is applied for instructions where 16-bit instruction code such as NOP Instruction or RET Instructions etc. is defined.

<table>
<thead>
<tr>
<th>MSB</th>
<th>OP</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0000</td>
</tr>
</tbody>
</table>

TYPE-F
It has 5-bit OP Code (OP) and 11-bit address designated filed (rel11).

<table>
<thead>
<tr>
<th>MSB</th>
<th>OP</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TYPE-G
It has 8-bit OP code (OP) and 20bit immediate data field (i20), register designated field (Ri). It has 32-bit length and is applied only for LDI:20 Instruction.

<table>
<thead>
<tr>
<th>MSB</th>
<th>OP</th>
<th>i20 (Upper)</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MSB</th>
<th>OP</th>
<th>i20 (Lower)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TYPE-H

It has 12-bit OP code (OP) and 32-bit immediate data field (i32), register designated field (Ri). It has 48-bit length and is applied only for LDI:32 Instruction.

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>OP</td>
</tr>
<tr>
<td>(n+2)</td>
<td>i32 (Upper)</td>
</tr>
<tr>
<td>(n+4)</td>
<td>i32 (Lower)</td>
</tr>
</tbody>
</table>

TYPE-I

It has 12-bit OP Code (OP) and 20-bit address designated filed (rel20). These are the instruction formats that have been added to FR81 Family CPU.

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>OP</td>
</tr>
<tr>
<td>(n+2)</td>
<td>rel20</td>
</tr>
</tbody>
</table>

TYPE-J

It has 12-bit OP Code (OP), register designated fields (Rj/FRi/cc) and 16-bit address designated fields (u16/rel16). These are the instruction formats that have been added to FR81 Family CPU.

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>OP</td>
</tr>
<tr>
<td>(n+2)</td>
<td>u16/rel16</td>
</tr>
</tbody>
</table>
### TYPE-K

It has 12-bit OP Code (OP), register designated field (Rj) and floating point register designated field (FRi). These are the instruction formats that have been added to FR81 Family CPU.

<table>
<thead>
<tr>
<th>MSB</th>
<th>OP</th>
<th>LSB</th>
<th>Rj</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(n+2)</td>
<td>-</td>
<td></td>
<td>FRi</td>
</tr>
</tbody>
</table>

### TYPE-L

It has 14-bit OP Code (OP) and 14-bit immediate data fields (o14/u14), floating point register designated field (FRi). Immediate data fields are not used in some instructions. These are the instruction formats that have been added to FR81 Family CPU.

<table>
<thead>
<tr>
<th>MSB</th>
<th>OP</th>
<th>LSB</th>
<th>o14/u14/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(n+2)</td>
<td>o14/u14/-</td>
<td></td>
<td>FRi</td>
</tr>
</tbody>
</table>

### TYPE-M

It has 16-bit OP Code (OP) and three floating point register designated fields (FRk, FRj, FRi). These are the instruction formats that have been added to FR81 Family CPU.

<table>
<thead>
<tr>
<th>MSB</th>
<th>OP</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(n+2)</td>
<td>-</td>
<td>FRk/-</td>
</tr>
</tbody>
</table>
TYPE-N

It has 14-bit OP Code (OP) and floating point register list (frlist). These are the instruction formats that have been added to FR81 Family CPU.

6.2.4 Register designated Field

- General-purpose register designated Field (Ri/Rj)

Among Instruction formats, fields that designate general-purpose register are 4-bit length Ri and Rj. Relation between bit pattern of general purpose register and register designated field has been indicated in Table 6.2-1.

<table>
<thead>
<tr>
<th>Ri / Rj</th>
<th>Register</th>
<th>Ri / Rj</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000_B</td>
<td>R0</td>
<td>1000_B</td>
<td>R8</td>
</tr>
<tr>
<td>0001_B</td>
<td>R1</td>
<td>1001_B</td>
<td>R9</td>
</tr>
<tr>
<td>0010_B</td>
<td>R2</td>
<td>1010_B</td>
<td>R10</td>
</tr>
<tr>
<td>0011_B</td>
<td>R3</td>
<td>1011_B</td>
<td>R11</td>
</tr>
<tr>
<td>0100_B</td>
<td>R4</td>
<td>1100_B</td>
<td>R12</td>
</tr>
<tr>
<td>0101_B</td>
<td>R5</td>
<td>1101_B</td>
<td>R13</td>
</tr>
<tr>
<td>0110_B</td>
<td>R6</td>
<td>1110_B</td>
<td>R14</td>
</tr>
<tr>
<td>0111_B</td>
<td>R7</td>
<td>1111_B</td>
<td>R15</td>
</tr>
</tbody>
</table>
6.2 Dedicated register designated Field (Rs)

Among instruction formats, field that designates dedicated register is 4-bit length Rs. Relation between bit pattern of dedicated register and register designated field has been indicated in Table 6.2-2.

Table 6.2-2 Bit pattern of dedicated register and register designated field

<table>
<thead>
<tr>
<th>Rs</th>
<th>Register</th>
<th>Rs</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000_B</td>
<td>Table Base Register (TBR)</td>
<td>1000_B</td>
<td>Exception status register (ESR)</td>
</tr>
<tr>
<td>0001_B</td>
<td>Return Pointer (RP)</td>
<td>1001_B</td>
<td></td>
</tr>
<tr>
<td>0010_B</td>
<td>System Stack Pointer (SSP)</td>
<td>1010_B</td>
<td></td>
</tr>
<tr>
<td>0011_B</td>
<td>User Stack Pointer (USP)</td>
<td>1011_B</td>
<td>Reserved</td>
</tr>
<tr>
<td>0100_B</td>
<td>Multiply/Divide Register (MDH)</td>
<td>1100_B</td>
<td></td>
</tr>
<tr>
<td>0101_B</td>
<td>Multiply/Divide Register (MDL)</td>
<td>1101_B</td>
<td></td>
</tr>
<tr>
<td>0110_B</td>
<td>Base pointer (BP)</td>
<td>1110_B</td>
<td></td>
</tr>
<tr>
<td>0111_B</td>
<td>FPU control register (FCR)</td>
<td>1111_B</td>
<td>Debug register (DBR)</td>
</tr>
</tbody>
</table>

Bit pattern which is shown as "Reserved" in the field that designates dedicated register is the reserved pattern. Operation when reserved pattern is specified is not covered by the warranty.

6.2 Floating point register designated field

Among instruction formats, fields that designate floating point register are 4-bit length FRi, FRj and FRk. The relationship between the bit pattern of the floating point register and register designated field is indicated in Table 6.2-3.

Table 6.2-3 Bit pattern of floating point register and register designated field

<table>
<thead>
<tr>
<th>FRk/FRj/FRi</th>
<th>Register</th>
<th>FRk/FRj/FRi</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000_B</td>
<td>FR0</td>
<td>1000_B</td>
<td>FR8</td>
</tr>
<tr>
<td>0001_B</td>
<td>FR1</td>
<td>1001_B</td>
<td>FR9</td>
</tr>
<tr>
<td>0010_B</td>
<td>FR2</td>
<td>1010_B</td>
<td>FR10</td>
</tr>
<tr>
<td>0011_B</td>
<td>FR3</td>
<td>1011_B</td>
<td>FR11</td>
</tr>
<tr>
<td>0100_B</td>
<td>FR4</td>
<td>1100_B</td>
<td>FR12</td>
</tr>
<tr>
<td>0101_B</td>
<td>FR5</td>
<td>1101_B</td>
<td>FR13</td>
</tr>
<tr>
<td>0110_B</td>
<td>FR6</td>
<td>1110_B</td>
<td>FR14</td>
</tr>
<tr>
<td>0111_B</td>
<td>FR7</td>
<td>1111_B</td>
<td>FR15</td>
</tr>
</tbody>
</table>
This section describes the data type and format supported by FR81 Family CPU. In addition to integer type supported by FR80 and earlier, single precision floating point type has been added.

### 6.3.1 Data Format Used by Integer Type Instructions (Common with All FR Family)

- **Signed integer byte**
  
  Signed integer byte is represented as consecutive 8 bits. Bit 7 represents the sign bit (S), and "0" represents positive or zero and "1" represents negative.
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>S</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Unsigned integer byte**
  
  Unsigned integer byte is represented as consecutive 8 bits.

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Signed integer half word**
  
  Signed integer half word is represented as consecutive 16 bits. Bit 15 represents the sign bit (S), and "0" represents positive or zero and "1" represents negative.

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>S</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Unsigned integer half word**
  
  Unsigned integer half word is represented as consecutive 16 bits.

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>
6.3

- **Signed integer word**

  Signed integer word is represented as consecutive 32 bits. Bit 31 represents the sign bit (S), and "0" represents positive or zero and "1" represents negative.

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>$S$</td>
<td></td>
</tr>
</tbody>
</table>

- **Unsigned integer word**

  Unsigned integer word is represented as consecutive 32 bits.

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>

### 6.3.2 Format Used for Floating Point Type Instructions

- **Floating point format**

  The IEEE 754 standard is used for floating point format. A floating point is represented by the following 3 fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Symbol</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign bit (Sign)</td>
<td>$s$</td>
<td>&quot;0&quot; represents positive, and &quot;1&quot; represents negative.</td>
</tr>
<tr>
<td>Exponent (Exponent)</td>
<td>$e$</td>
<td>Bias representation with single precision bias of 127, and the double precision bias of 1023</td>
</tr>
<tr>
<td>Fractional bit (Fraction)</td>
<td>$f$</td>
<td>The fraction represents a number less than 1, but the significant is 1 plus the fraction part.</td>
</tr>
</tbody>
</table>

  Using the above-described symbols, floating point (normalized number) is represented by the following formula. Bias representation with single precision bias is 127, and the double precision bias is 1023.

  $$(-1)^s \times 1.f \times 2^{(e - \text{bias})}$$

  In addition, there are special numbers of not-a-number (NaN), infinity ($\pm \infty$), zero and unnormalized number.

<table>
<thead>
<tr>
<th>Field</th>
<th>Symbol</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signaling NaN (SNaN)</td>
<td>$e$ - bias = $E_{\text{max}} + 1$, and MSB of $f$ is 0</td>
<td></td>
</tr>
<tr>
<td>Quiet NaN (QNaN)</td>
<td>$e$ - bias = $E_{\text{max}} + 1$, and MSB of $f$ is 1</td>
<td></td>
</tr>
<tr>
<td>Infinity ($+\infty$, $-\infty$)</td>
<td>$e$ - bias = $E_{\text{max}} + 1$, and $f$ is 0</td>
<td></td>
</tr>
<tr>
<td>Normalized number</td>
<td>$e$ - bias = Between $E_{\text{min}}$ and $E_{\text{max}}$</td>
<td></td>
</tr>
<tr>
<td>Unnormalized number</td>
<td>$e$ - bias = $E_{\text{min}} - 1$, and $f$ is not 0</td>
<td></td>
</tr>
<tr>
<td>Zero ($+0$, $-0$)</td>
<td>$e$, $f = 0$</td>
<td></td>
</tr>
</tbody>
</table>
Single precision floating point (32-bit)

This conforms to IEEE754 single precision format and is represented as consecutive 32 bits. In the single precision floating point format, bit 31 represents the sign bit (S), bit 30 to bit 23 represent the exponent bits, and bit 22 to bit 0 represent the fractional bits.
6.4 Read-Modify-Write type Instructions

Read-Modify-Write type Instructions are those that carry out a series of operations namely, arithmetic processing in the data read from the memory space and write the result in the same address of the memory space.

IN registers of peripheral functions (I/O Registers), there are bits whose read values are different depending on instructions that independently carry out read access like LD Instruction and Read-Modify-Write type Instructions. Such bits have been described in the explanation on registers (I/O Registers) of peripheral functions.

In case of Read-Modify-Write type Instructions, a different instruction based on EIT processing is not executed between Read access and Write access of one instruction. This is used for exclusive control that uses flag or semaphore between programs.

Whether or not an instruction is Read-Modify-Write system Instruction is defined for each instruction. See "CHAPTER 7 DETAILED EXECUTION INSTRUCTIONS".
6.5 Branching Instructions and Delay Slot

FR81 Family CPU Branching Instructions are of two types namely, Delayed Branching Instructions and Non-delayed Branching Instructions.

### 6.5.1 Delayed Branching Instructions

In case of Delayed Branching Instructions, prior to execution of Branching Destination Instructions, instructions immediately after Branching Instructions are executed. Instructions immediately after Delayed Branching Instructions are called Delay slot.

Branching Instructions having ".D" affixed to mnemonic are Delayed Branching Instructions. Next Instruction will be Delayed Branching Instruction.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Branching Instructions</th>
<th>Delayed Branching Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP:D @Ri</td>
<td>CALL:D label12</td>
<td>CALL:D @Ri RET:D</td>
</tr>
<tr>
<td>BRA:D label9</td>
<td>BNO:D label9</td>
<td>BEQ:D label9</td>
</tr>
<tr>
<td>BC:D label9</td>
<td>BNC:D label9</td>
<td>BN:D label9</td>
</tr>
<tr>
<td>BV:D label9</td>
<td>BNV:D label9</td>
<td>BLT:D label9</td>
</tr>
<tr>
<td>BLE:D label9</td>
<td>BGT:D label9</td>
<td>BLS:D label9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BHI:D label9</td>
</tr>
</tbody>
</table>

Since Delay Slot instructions are executed prior to Branching operation, apparent execution cycle of Branching Instruction will be 1 cycle. In case a valid instruction cannot be allocated in the Delay slot, it is necessary to allocate NOP Instruction. Example of Delayed Branching Instruction has been given below.

```
; Instructions alignment
ADD R1,R2
BRA:D LABEL ; Branching Instructions
MOV R2,R3 ; Delay slot (Executed before Branching)
...
LABEL: ST R3,@R4 ; Branching Destination
```

In case of Conditional Branching Instruction, whether or not Branching conditions are established, Delay slot instructions are executed.

Instructions that can be placed in the Delay slot are only those that satisfy following conditions. When an attempt is made to execute an instruction that cannot be placed in the delay slot, an invalid instruction exception occurs and the EIT processing is carried out.

- 1 cycle Instructions
- Those that are not Branching Instructions
• Instructions that do not affect the operation even when the order is changed

1 cycle Instructions are those where anyone variable namely 1, a, b, c, d is independently written in the CYC Column of "A.2 Instruction Lists". (Instructions which have 2a or 1+b are not 1 cycle instructions).

List of instructions that can be placed in Delay Slot are indicated in "Appendix A.3".

EIT processing such as Step Trace Trap, general interrupt, NMI etc. cannot be accepted between Delayed Branching Instructions and Delay Slot Instructions.

### 6.5.2 Specific example of Delayed Branching Instructions

Specific example of Delayed Branching Instruction is given below.

"JMP:D @Ri" Instruction, "CALL:D @Ri" Instruction

General-purpose register Ri referred to during JMP:D Instruction, CALL:D Instruction is not affected by the Branching Destination Address even if Delay Slot Instruction updates Ri.

[Ex]
```
LDI:32  #Label, R0
JMP:D   @R0        ; Branching in Label
LDI:8   #0, R0     ; Does not affect Branching Destination Address
...
```

RET:D Instruction

Return Pointer (RP) referred to by RET:D Instruction is not affected even if Delay Slot Instruction updates the Return Pointer (RP).

[Ex]
```
RET:D             ; Branching to address indicated by RP set prior to this
MOV     R8, RP    ; Not affected by Return Operation
...
```

Bcc:D Instructions

Flag of Condition Code Register (CCR) referred to by Bcc:D Instruction is not affected by Delay Slot Instructions.

[Ex]
```
ADD     #1, R0    ; Flag change
BC:D    overflow  ; Branching based on execution result of preceding ADD Instruction
ANDCCR  #0        ; Updating of this flag does not affect Branching
```
CALL:D Instruction

If Return Pointer (RP) is referred to based on Delay Slot Instruction of CALL:D Instruction, updated content will be read based on CALL:D Instruction.

[Ex]
CALL:D Label ; Branching after updating of RP
MOV RP,R0 ; Execution result of RP of preceding CALL:D Instruction is transferred to R0

6.5.3 Non-Delayed Branching Instructions

In case of Non-Delayed Branching Instructions, execution is carried out in the sequence of Instructions. Instruction immediately after Branching Instruction is never executed before branching.

Branching Instructions without ":D" in mnemonic are Non-Delayed Branching Instructions. Next instruction will be Non-Delayed Branching Instruction

JMP @Ri CALL label12 CALL @Ri RET
BRA label9 BNO label9 BEQ label9 BNE label9
BC label9 BNC label9 BN label9 BP label9
BV label9 BNV label9 BLT label9 BGE label9
BLE label9 BGT label9 BLS label9 BHI label9

Execution cycles of Non-Delayed Branching Instruction will be 2 cycles when Branching and 1 cycle when not branching. Example of Non-Delayed Branching Instruction is given below.

; Sequence of Instructions
ADD R1,R2
BRA LABEL ; Branching Instruction
MOV R2,R3 ; Not executed

LABEL: ST R3,@R4 ; Branching Destination

Compared to Delayed Branching Instructions where NOP Instruction is placed in the Delay Slot, efficiency of instruction code can be increased. Execution speed and Code Efficiency both can be realized by using Delayed Branching Instruction when valid instruction can be placed in the Delay Slot and using Non-delayed Branching Instruction otherwise.
6.6 Step Division Instructions

In FR81 Family CPU, 32-bit signed/unsigned division is carried out based on combination of Step Division Instructions.

Step Division Instructions are of following types.
- DIV0S (Initial Setting Up for Signed Division)
- DIV0U (Initial Setting Up for Unsigned Division)
- DIV1 (Main Process of Division)
- DIV2 (Correction When Remain is zero)
- DIV3 (Correction When Remain is zero)
- DIV4S (Correction Answer for Signed Division)

In order to realize signed division, combine the Instructions as follows.
DIV0S, DIV1 × 32, DIV2, DIV3, DIV4S

In order to realize unsigned division, combine the Instructions as follows.
DIV0U, DIV1 × 32

For various Instructions, see "CHAPTER 7 DETAILED EXECUTION INSTRUCTIONS".

6.6.1 Signed Division

Signed 32bit dividend is divided with signed 32 bit divisor and quotient of signed 32 bit and remainder of signed 32bit are obtained.

Before carrying out division, dividend and divisor are set in the following register.
- Multiplication/Division Register (MDL): Dividend of signed 32 bit (Dividend)
- One of general-purpose registers: Divisor of signed 32 bit (Divisor)

Signed division is carried out by executing following 36 Instructions. DIV1 Instructions 32 numbers are arranged after DIV0S Instructions. In the operand of DIV0S Instructions, DIV1 Instructions, DIV2 Instructions general-purpose registers that store divisor are specified.

DIV0S R2 ; Divisor in R2
DIV1 R2 ; #1
DIV1 R2 ; #2
... 
DIV1 R2 ; #30
Division results are stored in the following registers.
- Multiplication/Division Register (MDL): quotient of signed 32 bit
- Multiplication/Division Register (MDH): remainder of signed 32 bit

Example of execution of signed division has been indicated in Figure 6.6-1.

6.6.2 Unsigned Division

Dividend of unsigned 32 bit dividend is divided with unsigned 32bit divisor and quotient of unsigned 32 bit and remainder of unsigned 32 bit are obtained.

Before carrying out division, dividend and divisor are set in the following register.
- Multiplication/Division Register (MDL): Dividend of unsigned 32 bit (Dividend)
- One of general-purpose registers: Divisor of unsigned 32 bit (Divisor)

Unsigned division is carried out by executing following 33 Instructions. DIV1 Instructions 32 numbers are arranged after DIV0U Instruction. In the operand of DIV0U Instructions, DIV1 Instructions, general-purpose registers that store divisor are specified.
DIV0S R2 ; divisor in R2
DIV1 R2 ; #1
DIV1 R2 ; #2

... 
DIV1 R2 ; #30
DIV1 R2 ; #31
DIV1 R2 ; #32

Division result is stored in the following registers.
- Multiplication/Division Register (MDL): quotient of unsigned 32 bit
- Multiplication/Division Register (MDH): remainder of unsigned 32 bit

Example of execution of unsigned division has been indicated in Figure 6.6-2.

**Figure 6.6-2 Example of execution of unsigned division**

<table>
<thead>
<tr>
<th>R2</th>
<th>MDH</th>
<th>MDL</th>
<th>SCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>xxxxxxxx</td>
<td>F E D C B A 9 8</td>
<td>x x 0</td>
</tr>
<tr>
<td>R2</td>
<td>MDH</td>
<td>MDL</td>
<td>SCR</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>0 0 0 0 0 0 7 8</td>
<td>0 0 0 0 0 0 E 0</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>

Before execution | After execution

In SOFTUNE Assembler, DIVU Instruction has been arranged to carry out unsigned division as Assembler Pseudo Machine Instruction. Using this DIVU Instruction in place of above mentioned 33 Instructions, unsigned division can be described with 1 Instruction. See "FR family SOFTUNE Assembler Manual" for DIVU Instruction.
CHAPTER 7

DETAILED EXECUTION

INSTRUCTIONS

This chapter explains each of the execution instructions used by the FR81 Family CPU, alphabetically in the reference format.
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

Refer to "A.1 Meaning of Symbols" for explanation regarding symbols used in detailed execution instructions. The respective instructions are explained separately in the following items.

- **Assembler Format**
  Shows the format of writing the instruction in the assembler language.

- **Operation**
  Shows the operation of an instruction by substituting it with an arrow mark (→).

- **Flag Change**
  Shows whether the flag of the Condition Code Register (CCR) changes by the execution of an instruction.

- **Classification**
  Shows Functional classification of instructions and the following sections of instructions.
  Instruction with delay slot: Instruction than can be positioned in the delay slot
  Read-Modify-Write system Instruction
  FR80 Family: Instructions added to FR80 Family and after CPUs
  FR81 Family: Instructions added in FR81 Family CPU
  FR81 Updating: Instruction to which definition is changed in FR81 family CPU

- **Execution Cycle**
  Shows the required number of clock cycles for instruction execution

- **Instruction Format**
  Shows the format and bit pattern of the instruction.

- **Execution example**
  Shows the operation example at the time of instruction execution.
7.1 ADD (Add 4bit Immediate Data to Destination Register)

Adds the result of higher 28 bits of 4-bit immediate data with zero extension(0-15) and stores the results to Ri.

- **Assembler Format**
  
  ```
  ADD   #i4, Ri
  ```

- **Operation**
  
  ```
  Ri + extu(i4) → Ri
  ```

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

  - N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  - Z: Set when the operation result is zero, cleared otherwise.
  - V: Set when an overflow has occurred as a result of the operation, cleared otherwise.
  - C: Set when a carry has occurred as a result of the operation, cleared otherwise.

- **Classification**
  
  Add/Subtract Instruction, Instruction with delay slot

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0</td>
<td>i4 Ri</td>
</tr>
</tbody>
</table>
7.1 Execution Example

ADD #2, R3 ; Bit pattern of instruction: 1010 0100 0010 0011

Before execution

<table>
<thead>
<tr>
<th>R3</th>
<th>9 9 9 9 9 9 9 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Z V C</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R3</th>
<th>9 9 9 9 9 9 9 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Z V C</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>CCR</td>
<td>1 0 0 0</td>
</tr>
</tbody>
</table>
7.2 ADD (Add Word Data of Source Register to Destination Register)

Adds word data of Rj to Ri, stores result to Ri.

- **Assembler Format**
  
  \[
  \text{ADD } R_j, R_i
  \]

- **Operation**
  
  \[
  R_i + R_j \rightarrow R_i
  \]

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

  - N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  - Z: Set when the operation result is zero, cleared otherwise.
  - V: Set when an overflow has occurred as a result of the operation, cleared otherwise.
  - C: Set when a carry has occurred as a result of the operation, cleared otherwise.

- **Classification**

  Add/Subtract Instruction, Instruction with delay slot

- **Execution Cycles**

  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>Rj</td>
</tr>
<tr>
<td>Ri</td>
</tr>
<tr>
<td>LSB</td>
</tr>
</tbody>
</table>

  \[
  \text{NZVC}
  \]
  \[
  \text{CCCC}
  \]
  \[
  \text{10100110}
  \]

  \[
  \text{Rj}
  \]
  \[
  \text{Ri}
  \]
### Execution Example

ADD R2, R3 ; Bit pattern of instruction: 1010 0110 0010 0011

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>8 7 6 5 4 3 2 1</td>
</tr>
</tbody>
</table>

NZVC CCR: 0 0 0 0

Before execution

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>9 9 9 9 9 9 9</td>
</tr>
</tbody>
</table>

NZVC CCR: 1 0 0 0

After execution
7.3 ADD2 (Add 4bit Immediate Data to Destination Register)

Adds the result of the higher 28 bits of 4-bit immediate data with minus extension (-16 to -1) to word data in Ri, stores results to Ri. Unlike SUB instruction, changing C flag of this instruction becomes it as well as the ADD instruction unlike the SUB instruction.

- Assembler Format
  ADD2 #i4, Ri

- Operation
  Ri + extn(i4) → Ri

- Flag Change
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V: Set when an overflow has occurred as a result of the operation, cleared otherwise.
  C: Set when a carry has occurred as a result of the operation, cleared otherwise.

- Classification
  Add/Subtract Instruction, Instruction with delay slot

- Execution Cycles
  1 cycle

- Instruction Format
  MSB              LSB
  | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | i4 | Ri |
Execution Example

ADD2   #-2, R3  ; Bit pattern of instruction: 1010 0101 1110 0011

Before execution  After execution

R3  | 9 9 9 9 9 9 9 9 9 9| R3  | 9 9 9 9 9 9 9 9 7|
NZVC | N Z V C | N Z V C |
CCR  | 0 0 0 0  | CCR  | 1 0 0 1  |

Before execution  After execution
# ADDC (Add Word Data of Source Register and Carry Bit to Destination Register)

Adds word data and carry flag (C) of Rj to Ri, stores results in Ri.

- **Assembler Format**
  
  ```
  ADDC Rj, Ri
  ```

- **Operation**
  
  ```
  Ri + Rj + C → Ri
  ```

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

  - N: Set when MSB of the operation result is "1", cleared when MSB is "0".
  - Z: Set when the operation result is zero, cleared otherwise.
  - V: Set when an overflow has occurred as a result of the operation, cleared otherwise.
  - C: Set when a carry has occurred as a result of the operation, cleared otherwise.

- **Classification**

  Add/Subtract Instruction, Instruction with delay slot

- **Execution Cycles**

  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 1 1</td>
<td>Rj</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.4 Execution Example

```
ADDC R2, R3 ; Bit pattern of the instruction: 1010 0111 0010 0011
```

Before execution After execution

<table>
<thead>
<tr>
<th>R2</th>
<th>1 2 3 4 5 6 7 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>8 7 6 5 4 3 2 0</td>
</tr>
</tbody>
</table>

```
NZVC CCR 0 0 0 1
```

```
NZVC CCR 1 0 0 0
```

Before execution After execution
ADDN (Add Immediate Data to Destination Register)

Adds the result of the higher 28 bits of the 4-bit immediate data with zero extension (0 to 15) to the word data of Ri, stores the results without changing flag settings.

- Assembler Format
  
  ADDN  #i4, Ri

- Operation
  
  Ri + extu(i4) → Ri

- Flag Change
  
  N, Z, V, C: Unchanged.

- Classification
  
  Add/Subtract Instruction, Instruction with delay slot

- Execution Cycles
  
  1 cycle

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>i4</th>
<th>Ri</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

  ```
### Execution Example

**ADDN #2, R3**  
Bit pattern of the instruction: 1010 0000 0010 0011

<table>
<thead>
<tr>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3 9999 9979 ..</td>
<td>R3 9999 9979 ..</td>
</tr>
<tr>
<td>N Z V C 0000</td>
<td>N Z V C 0000</td>
</tr>
</tbody>
</table>

Before execution: R3 = 9999 9979, CCR = 0000

After execution: R3 = 9999 9979, CCR = 0000
7.6 ADDN (Add Word Data of Source Register to Destination Register)

Adds the word data of Rj to the word data of Ri, stores results in Ri without changing flag settings.

- **Assembler Format**
  
  ADDN Rj, Ri

- **Operation**
  
  Ri + Rj → Ri

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**

  Add/Subtract Instruction, Instruction with delay slot

- **Execution Cycles**

  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>Rj</th>
<th>Ri</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Execution Example

ADDN R2, R3 ; Bit pattern of the instruction: 1010 0010 0010 0011

Before execution  After execution

R2 | R3 | N Z V C | R2 | R3 | N Z V C
---|----|-------|---|----|-------
1 2 3 4 5 6 7 8 | 8 7 6 5 4 3 2 1 | 0 0 0 0 | 9 9 9 9 9 9 9 9 | 0 0 0 0

CCR

Before execution

After execution
7.7 ADDN2 (Add Immediate Data to Destination Register)

Adds the result of the higher 28 bits of 4-bit immediate data with minus extension (-16 to -1) to word data in Ri, stores the results in Ri without changing flag settings.

- **Assembler Format**
  
  \[
  \text{ADDN2} \ #i4, \ Ri
  \]

- **Operation**
  
  \[
  \text{Ri} + \text{extn}(i4) \rightarrow \text{Ri}
  \]

- **Flag Change**
  
  \[
  \begin{array}{cccc}
  \text{N} & \text{Z} & \text{V} & \text{C} \\
  - & - & - & - \\
  \end{array}
  \]

  N, Z, V, C: Unchanged.

- **Classification**
  
  Add/Subtract Instruction, Instruction with delay slot

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
  \[
  \begin{array}{cccccc}
  \text{MSB} & 1 & 0 & 1 & 0 & 0 & 0 & 1 & \text{i4} & \text{Ri} \\
  \text{LSB} & & & & & & & & & \\
  \end{array}
  \]
Execution Example

ADDN2  #-2, R3 ; Bit pattern of the instruction: 1010 0001 1110 0011

9 9 9 9 9 9 9 9 9 9 9 7
R3
N Z V C

0 0 0 0
CCR

Before execution

9 9 9 9 9 9 9 9 7
R3
N Z V C

0 0 0 0
CCR

After execution
7.8 ADDSP (Add Stack Pointer and Immediate Data)

Adds 4 times the 8-bit immediate data as a signed extended value to the word data of R15 and stores result in R15. Specifies the value of $s8 \times 14$ as $s10$.

- **Assembler Format**
  ```assembly```
  ADDSP  #s10
  ```

- **Operation**
  ```
  R15 + exts(s8x4) → R15
  ```

- **Flag Change**
  ```
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
  ```
  N, Z, V, C: Unchanged.

- **Classification**
  Other instructions, Instruction with delay slot

- **Execution Cycles**
  1 cycle

- **Instruction Format**
  ```
  MSB | LSB
  ----|----
  1   | 0  
  1   | 0  
  0   | 0  
  1   | 1  
  1   | s8 |
  ```
Execution Example

ADDSP #4 ; Bit pattern of the instruction: 1010 0011 1111 1111

Before execution

R15

8 0 0 0 0 0 0 0

After execution

R15

7 F F F F F F C
FR81 Family

7.9 AND (And Word Data of Source Register to Data in Memory)

Takes the logical AND of the word data at memory address Ri and word data in Rj and stores the results to the memory address corresponding to Ri.

- **Assembler Format**
  
  `AND Rj,@Ri`

- **Operation**
  
  `(Ri) & Rj → (Ri)`

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  - N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  - Z: Set when the operation result is zero, cleared otherwise.
  - V, C: Unchanged.

- **Classification**
  
  Logical calculation instruction, Read/Modify/Write type instruction

- **Execution Cycle**
  
  1+2a cycles

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rj</td>
<td>Ri</td>
</tr>
</tbody>
</table>
Execution Example

`AND  R2,@R3   ; Bit pattern of the instruction: 1000 0100 0010 0011`

Before execution

<table>
<thead>
<tr>
<th>R2</th>
<th>1 1 1 1 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>1234567C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N Z V C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCR</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R2</th>
<th>1 1 1 1 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>1234567C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N Z V C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCR</td>
</tr>
</tbody>
</table>
7.10  AND (And Word Data of Source Register to Destination Register)

Takes the logical AND of word data in Ri and word data in Rj and stores the results to Rj.

- **Assembler Format**
  
  \[ \text{AND} \quad Rj, Ri \]

- **Operation**
  
  \[ Ri \& Rj \rightarrow Ri \]

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  - N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  - Z: Set when the operation result is zero, cleared otherwise.
  - V, C: Unchanged.

- **Classification**
  
  Logical calculation instruction, Instruction with delay slot

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
  MSB | LSB
  ---|---
  1 0 0 0 0 0 1 0 | Rj  Ri
7.10 Execution Example

AND R2, R3 ; Bit pattern of the instruction: 1000 0010 0010 0011

Before execution          After execution

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 0000</td>
<td>1010 1010</td>
<td>1111 0000</td>
<td>1010 0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NZVC</th>
<th>NZVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

Before execution

After execution
7.11 ANDB (And Byte Data of Source Register to Data in Memory)

Takes the logical AND of the byte data at memory address Ri and the byte data in Rj and stores the results at Ri location in the memory.

- Assembler Format
  \[
  \text{ANDB} \ Rj, @Ri
  \]

- Operation
  \[
  (Ri) \& \ Rj \rightarrow (Ri)
  \]

- Flag Change
  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  C & C & - & -
  \end{array}
  \]
  
  - N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  - Z: Set when the operation result is zero, cleared otherwise.
  - V, C: Unchanged.

- Classification
  Logical calculation instruction, Read/Modify/Write type instruction

- Execution Cycles
  \[1+2a \text{ cycles}\]

- Instruction Format
  \[
  \begin{array}{ccccccccc}
  \text{MSB} & & & & & & & & \\
  1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \text{Rj} & \text{Ri}
  \end{array}
  \]
Execution Example

ANDB R2, @R3 ; Bit pattern of the instruction: 1000 0110 0010 0011

Before execution

<table>
<thead>
<tr>
<th>R2</th>
<th>0 0 0 0 0 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>Memory 12345678</td>
<td>1 1</td>
</tr>
<tr>
<td>Memory 12345679</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R2</th>
<th>0 0 0 0 0 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>Memory 12345678</td>
<td>1 0</td>
</tr>
<tr>
<td>Memory 12345679</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>
7.12 ANDCCR (And Condition Code Register and Immediate Data)

Takes the logical AND of byte data in the condition code register (CCR) and 8-bit immediate data and returns the results to the CCR.

- Assembler Format
  
  ```
  ANDCCR   #u8
  ```

- Operation
  
  User mode:
  
  ```
  CCR & (u8 | 30H) → CCR
  ```

  Privilege mode
  
  ```
  CCR & u8 → CCR
  ```

  In user mode, a request to rewrite the stack flag (S) or the interrupt enable flag (I) is ignored. The S and I flags can only be changed in privilege mode.

- Flag Change
  
<table>
<thead>
<tr>
<th>S</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

  S, I, N, Z, V, C: Varies according to results of operation.

- Classification
  
  Other instructions, Instruction with delay slot, FR81 updating

- Execution Cycles
  
  1 cycle

- Instruction Format
  
  MSB  | 1 0 0 0 0 0 1 1 | u8
  LSB  |  |  |  |  |  |  |  |  |  |  |
EIT Occurrence and Detection

An interrupt is detected (the value of I flag after instruction execution is used).

Execution Example

ANDCCR #0FEH ; Bit pattern of the instruction: 1000 0011 1111 1110

<table>
<thead>
<tr>
<th>S I N Z V C</th>
<th>S I N Z V C</th>
</tr>
</thead>
<tbody>
<tr>
<td>☐ ☐ ☐ ☑ ☐</td>
<td>☐ ☐ ☐ ☑ ☐</td>
</tr>
</tbody>
</table>

Before execution | After execution
FR81 Family

7.13 ANDH (And Halfword Data of Source Register to Data in Memory)

Takes the logical AND of the half-word data at Ri location of the memory and the half-word data in Rj and stores the results at Ri location of the memory.

- **Assembler Format**
  
  \[ \text{ANDH} \quad \text{Rj, @Ri} \]

- **Operation**
  
  \[(\text{Ri}) \& \text{Rj} \rightarrow (\text{Ri})\]

- **Flag Change**
  
  \[
  \begin{array}{cccc}
  \text{N} & \text{Z} & \text{V} & \text{C} \\
  \text{C} & \text{C} & - & - \\
  \end{array}
  \]

  N: Set when the MSB of the operation result is "1", cleared when the MSB (bit15) is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V, C: Unchanged.

- **Classification**
  
  Logical calculation instruction, Read/Modify/Write type instruction

- **Execution Cycles**
  
  1+2a cycles

- **Instruction Format**
  
  \[
  \begin{array}{ccccccc}
  \text{MSB} & & & & & & \text{LSB} \\
  1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \text{Rj} & \text{Ri} \\
  \end{array}
  \]
Execution Example

ANDH R2, @R3 ; Bit pattern of the instruction: 1000 0101 0010 0011

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
<th>Memory</th>
<th>R2</th>
<th>R3</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1100</td>
<td>1234 5678</td>
<td>1 0 1 0</td>
<td>0000 1100</td>
<td>1234 5678</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>1234567A</td>
<td>N Z V C</td>
<td>1234567A</td>
<td>N Z V C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
<td>CCR</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before execution | After execution
7.14  ASR (Arithmetic shift to the Right Direction)

Makes an arithmetic right shift of the word data in Ri by Rj bits, stores the result to Ri. Only the lower 5 bits of Rj, which designates the size of the shift, are valid and the shift range is 0 to 31 bits.

- **Assembler Format**
  
  ASR   Rj, Ri

- **Operation**
  
  Ri >> Rj → Ri

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>C</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V: Unchanged.
  C: Holds the bit value shifted last. Cleared when the shift amount is "0".

- **Classification**
  
  Shift instructions, Instruction with delayed slot

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>Rj</td>
</tr>
<tr>
<td>Ri</td>
<td></td>
</tr>
</tbody>
</table>
EXECUTION EXAMPLE

ASR R2, R3 ; Bit pattern of the instruction: 1011 1010 0010 0011

| R2  | 0000 0000 0000 | R2  | 0000 0000 0000 |
| R3  | FFFFFFFF FFFF  | R3  | FFFFFFFF 0FFF  |
| CCR | 0000 0000     | CCR | 1001 0001     |

Before execution  After execution
7.15 ASR (Arithmetic shift to the Right Direction)

Makes an arithmetic right shift of the word data in Ri by u4 bits, stores the result to Ri.

- **Assembler Format**
  
  ASR #u4, Ri

- **Operation**
  
  Ri >> u4 → Ri

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>C</td>
</tr>
</tbody>
</table>

  - N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  - Z: Set when the operation result is zero, cleared otherwise.
  - V: Unchanged.
  - C: Holds the bit value shifted last. Cleared when the shift amount is "0".

- **Classification**

  Shift instruction, Instruction with delay slot

- **Execution Cycles**

  1 cycle

- **Instruction Format**

  MSB

  | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | u4 | Ri |

  LSB
Chapter 7 Detailed Execution Instructions

7.15 Execution Example

ASR #8, R3 ; Bit pattern of the instruction: 1011 1000 1000 0011

Before execution

<table>
<thead>
<tr>
<th>R3</th>
<th>FF0F</th>
<th>FFFF</th>
<th>NZVC</th>
<th>CCR</th>
<th>0000</th>
</tr>
</thead>
</table>

After execution

<table>
<thead>
<tr>
<th>R3</th>
<th>FFF0</th>
<th>FFFF</th>
<th>NZVC</th>
<th>CCR</th>
<th>1001</th>
</tr>
</thead>
</table>

Before execution | After execution
7.16 ASR2 (Arithmetic shift to the Right Direction)

Makes an arithmetic right shift of the word data in Ri by u4+16 bits, stores the result to Ri.

- **Assembler Format**
  
  ASR2 #u4, Ri

- **Operation**

  Ri >> {u4 + 16} → Ri

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>C</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V: Unchanged.
  C: Holds the bit value shifted last.

- **Classification**

  Shift instruction, Instruction with delayed slot

- **Execution Cycles**

  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>u4</td>
<td>Ri</td>
</tr>
</tbody>
</table>
7.16 Execution Example

ASR2  #8, R3  ; Bit pattern of the instruction: 1011 1001 1000 0011

Before execution

<table>
<thead>
<tr>
<th>R3</th>
<th>CCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>F 0 F F F F F F</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>N Z V C</td>
<td></td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R3</th>
<th>CCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>F F F F F F F 0</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>N Z V C</td>
<td></td>
</tr>
</tbody>
</table>
7.17  BANDH (And 4bit Immediate Data to Higher 4bit of Byte Data in Memory)

Takes the logical AND of the 4-bit immediate data and the higher 4 bits of byte data at memory Ri, stores the results to the memory address corresponding to Ri.

- Assembler Format
  
  BANDH  #u4,@Ri

- Operation
  
  \((Ri) \& \{u4 << 4 + 0F_{H}\} \rightarrow (Ri)\)  [Operation uses higher 4 bits only]

- Flag Change
  
  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  - & - & - & - \\
  \end{array}
  \]

  N, Z, V, C: Unchanged.

- Classification
  
  Bit operation instruction, Read/Modify/Write type instruction

- Execution Cycles
  
  1+2a cycles

- Instruction Format
  
  \[
  \begin{array}{cccccccc}
  \text{MSB} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \text{u4} & \text{Ri} \\
  \text{LSB} & & & & & & & & & & \\
  \end{array}
  \]
Execution Example

BANDH #0,@R3 ; Bit pattern of the instruction: 1000 0001 0000 0011

Before execution

<table>
<thead>
<tr>
<th>R3</th>
<th>Memory</th>
<th>CCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>1 1</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>12345679</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R3</th>
<th>Memory</th>
<th>CCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>0 1</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>12345679</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.18 BANDL (And 4bit Immediate Data to Lower 4bit of Byte Data in Memory)

Takes the logical AND of the 4-bit immediate data and the lower 4 bits of byte data at memory Ri, stores the results to the memory address corresponding to Ri.

- **Assembler Format**
  
  BANDL   #u4,@Ri

- **Operation**
  
  
  \[(Ri) \& \{\text{F0H} + u4\} \rightarrow (Ri)\]  
  [Operation uses lower 4 bits only]

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**

  Bit operation instructions, Read/Modify/Write type instruction

- **Execution Cycles**

  1+2a cycles

- **Instruction Format**

  MSB                  LSB
  | 1 | 0 | 0 | 0 | 0 | 0 | 0 | u4 | Ri |
7.18 Execution Example

BANDL #0, @R3 ; Bit pattern of the instruction: 1000 0000 0000 0011

<table>
<thead>
<tr>
<th>R3</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345679</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12345679</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N Z V C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Before execution

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>12345679</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>12345679</td>
</tr>
</tbody>
</table>

NZVC
CCR0000

before execution

After execution
7.19 Bcc (Branch relative if Condition satisfied)

This is a branching instruction without a delay slot. If the conditions specified for each instruction are satisfied, branch to the address indicated by label9 relative to the value of the program counter (PC). When calculating the address, double the value of rel8 as a signed extension. If conditions are not satisfied, no branching occurs.

● Assembler Format

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>cc</th>
<th>Condition</th>
<th>Mnemonic</th>
<th>cc</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRA</td>
<td>0000</td>
<td>Always satisfied</td>
<td>BV</td>
<td>1000</td>
<td>V == 1</td>
</tr>
<tr>
<td>BNO</td>
<td>0001</td>
<td>Always unsatisfied</td>
<td>BNV</td>
<td>1001</td>
<td>V == 0</td>
</tr>
<tr>
<td>BEQ</td>
<td>0010</td>
<td>Z == 1</td>
<td>BLT</td>
<td>1010</td>
<td>(V ^ N) == 1</td>
</tr>
<tr>
<td>BNE</td>
<td>0011</td>
<td>Z == 0</td>
<td>BGE</td>
<td>1011</td>
<td>(V ^ N) == 0</td>
</tr>
<tr>
<td>BC</td>
<td>0100</td>
<td>C == 1</td>
<td>BLE</td>
<td>1100</td>
<td>((V ^ N)</td>
</tr>
<tr>
<td>BNC</td>
<td>0101</td>
<td>C == 0</td>
<td>BGT</td>
<td>1101</td>
<td>((V ^ N)</td>
</tr>
<tr>
<td>BN</td>
<td>0110</td>
<td>N == 1</td>
<td>BLS</td>
<td>1110</td>
<td>(C</td>
</tr>
<tr>
<td>BP</td>
<td>0111</td>
<td>N == 0</td>
<td>BHI</td>
<td>1111</td>
<td>(C</td>
</tr>
</tbody>
</table>

*: Logical add (or) ^: Exclusive-OR (exor) ===: comparison operation (satisfied by congruence)

Branching of each instruction is shown in Table 7.19-1.
7.19

● Flag Change

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

N, Z, V, C: Unchanged.

● Classification

Non-delayed branching instruction

● Execution Cycles

At time of branching: 2 cycles
At the time of no branching: 1 cycle

● Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>cc</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rel8</td>
</tr>
</tbody>
</table>

● Execution Example

BHI label ; Bit pattern of the instruction: 1110 1111 0010 1000

... label: ; Address of BHI Instruction + 50H

Before execution

PC | FF 80 00 00 00
N Z V C | 1 0 1 0
CCR | 1 0 1 0

After execution

PC | FF 80 00 05 2
N Z V C | 1 0 1 0
CCR | 1 0 1 0

Z or C = 0 (conditions satisfied)
This is a branching instruction with a delay slot. If the conditions established for each particular instruction are satisfied, branch to the address indicated by label9 relative to the value of the program counter (PC). When calculating the address, double the value of rel8 as a signed extension. If conditions are not satisfied, no branching occurs.

**Assembler Format**

- BRA:D label9
- BNO:D label9
- BEQ:D label9
- BNE:D label9
- BNC:D label9
- BN:D label9
- BP:D label9
- BV:D label9
- BNV:D label9
- BLT:D label9
- BGED label9
- BLE:D label9
- BGT:D label9
- BLD label9
- BHI:D label9

**Operation**

if (condition) then

PC + 2 + exts(rel8 × 2) → PC

Branching conditions of each instruction are shown in Table 7.20-1.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>cc</th>
<th>Condition</th>
<th>Mnemonic</th>
<th>cc</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRA:D</td>
<td>0000</td>
<td>Always satisfied</td>
<td>BV:D</td>
<td>1000</td>
<td>V == 1</td>
</tr>
<tr>
<td>BNO:D</td>
<td>0001</td>
<td>Always unsatisfied</td>
<td>BNV:D</td>
<td>1001</td>
<td>V == 0</td>
</tr>
<tr>
<td>BEQ:D</td>
<td>0010</td>
<td>Z == 1</td>
<td>BLT:D</td>
<td>1010</td>
<td>(V ^ N) == 1</td>
</tr>
<tr>
<td>BNE:D</td>
<td>0011</td>
<td>Z == 0</td>
<td>BGE:D</td>
<td>1011</td>
<td>(V ^ N) == 0</td>
</tr>
<tr>
<td>BC:D</td>
<td>0100</td>
<td>C == 1</td>
<td>BLE:D</td>
<td>1100</td>
<td>((V ^ N)</td>
</tr>
<tr>
<td>BNC:D</td>
<td>0101</td>
<td>C == 0</td>
<td>BGT:D</td>
<td>1101</td>
<td>((V ^ N)</td>
</tr>
<tr>
<td>BN:D</td>
<td>0110</td>
<td>N == 1</td>
<td>BLS:D</td>
<td>1110</td>
<td>(C</td>
</tr>
<tr>
<td>BP:D</td>
<td>0111</td>
<td>N == 0</td>
<td>BHI:D</td>
<td>1111</td>
<td>(C</td>
</tr>
</tbody>
</table>

*: Logical add (or) ^: Exclusive-OR (exor) ==: comparison operation (satisfied by congruence)
7.20

● Flag Change

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

N, Z, V, C: Unchanged.

● Classification

Delayed branching instruction

● Execution Cycles

1 cycle

● Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>cc</td>
<td>rel8</td>
</tr>
</tbody>
</table>

● Execution Example

BHI:D label ; Bit pattern of the instruction: 1111 1111 0010 1000
LDI:8 #255, R1 ; Instruction placed in delay slot
...
label: ; BHI:D instruction address + 50H

<table>
<thead>
<tr>
<th>R1</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 9 4 7 9 7 A F</td>
<td>0 0 0 0 0 0 F F</td>
</tr>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>F F 8 0 0 0 0 0</td>
<td>F F 8 0 0 0 5 2</td>
</tr>
<tr>
<td>N Z V C</td>
<td>N Z V C</td>
</tr>
<tr>
<td>CCR</td>
<td>CCR</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>1 0 1 0</td>
</tr>
</tbody>
</table>

Z or C = 0, conditions satisfied

Before execution After execution

The instruction placed in delay slot will be executed before the execution of the branch destination instruction. The value of R1 above will vary according to the specifications of the LDI:8 instruction placed in the delay slot.
7.21 BEORH (Eor 4bit Immediate Data to Higher 4bit of Byte Data in Memory)

Takes the logical exclusive OR of the 4-bit immediate data and the higher 4 bits of byte data at memory address Ri, stores the results to the memory address corresponding to Ri.

- Assembler Format
  
  BEORH #u4,@Ri

- Operation

  \((Ri) \oplus (u4 << 4) \rightarrow (Ri)\)  \[Operation uses higher 4 bits only\]

- Flag Change

  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  - & - & - & - \\
  \end{array}
  \]

  N, Z, V, C: Unchanged.

- Classification

  Bit Operation instruction, Read/Modify/Write type instruction

- Execution Cycles

  1+2a cycles

- Instruction Format

  \[
  \begin{array}{cccccccc}
  \text{MSB} & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & u4 & \text{Ri} \\
  \end{array}
  \]
 Execution Example

BEORH #1,@R3 ; Bit pattern of the instruction: 1001 1001 0001 0011

<table>
<thead>
<tr>
<th>R3</th>
<th>1 2 3 4 5 6 7 8</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>12345679</td>
<td>0 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R3</th>
<th>1 2 3 4 5 6 7 8</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>12345679</td>
<td>0 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CCR</th>
<th>0 0 0 0</th>
</tr>
</thead>
</table>

Before execution

After execution
FR81 Family

7.22 BEORL (Eor 4bit Immediate Data to Lower 4bit of Byte Data in Memory)

Takes the logical exclusive OR of the 4-bit immediate data and the lower 4 bits of byte data at memory address Ri, stores the results to the memory address corresponding to Ri.

- **Assembler Format**
  
  BEORL #u4,@Ri

- **Operation**
  
  \[(\text{Ri}) \ ^\lor \ u4 \rightarrow \ (\text{Ri})\]  
  [Operation uses lower 4 bits only]

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Bit Operation instruction, Read/Modify/Write type instruction

- **Execution Cycles**
  
  1+2a cycles

- **Instruction Format**
  
  \[
  \begin{array}{c|c|c|c|c|c|c|c|c|c|c}
  \hline
  \text{MSB} & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & u4 & \text{Ri} \\
  \hline
  \end{array}
  \]
### 7.22 Execution Example

BEORL #1,R3 ; Bit pattern of the instruction: 1001 1000 0001 0011

<table>
<thead>
<tr>
<th>R3</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>12345678</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12345679</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NZVC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before execution | After execution
BORH (Or 4bit Immediate Data to Higher 4bit of Byte Data in Memory)

Takes the logical OR of the 4-bit immediate data and the higher 4 bits of byte data at memory address Ri, stores the results to the memory address corresponding to Ri.

- **Assembler Format**
  
  BORH #u4, @Ri

- **Operation**
  
  \[(Ri) \| \{u4 \ll 4\} \rightarrow (Ri)\]  
  [Operation uses higher 4 bits only]

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Bit Operation instruction, Read/Modify/Write type instruction

- **Execution Cycles**
  
  1+2a cycles

- **Instruction Format**
  
  \[
  \begin{array}{cccccccc}
  \text{MSB} & 1 & 0 & 0 & 1 & 0 & 0 & 1 & \text{u4} & \text{Ri} \\
  \text{LSB} & & & & & & & & & \\
  \end{array}
  \]
 Execution Example

BORH #1,@R3 ; Bit pattern of the instruction: 1001 0001 0001 0011

Before execution

<table>
<thead>
<tr>
<th>R3</th>
<th>1 2 3 4 5 6 7 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>0 0</td>
</tr>
<tr>
<td>12345679</td>
<td></td>
</tr>
<tr>
<td>N Z V C</td>
<td></td>
</tr>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R3</th>
<th>1 2 3 4 5 6 7 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>1 0</td>
</tr>
<tr>
<td>12345679</td>
<td></td>
</tr>
<tr>
<td>N Z V C</td>
<td></td>
</tr>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>
**7.24  BORL (Or 4bit Immediate Data to Lower 4bit of Byte Data in Memory)**

Takes the logical OR of the 4-bit immediate data and the lower 4 bits of byte data at memory address Ri, stores the results to the memory address corresponding to Ri.

- **Assembler Format**
  
  \[ \text{BORL} \ #u4,@Ri \]

- **Operation**
  
  \[(Ri) \ | \ u4 \rightarrow (Ri) \] [Operation uses lower 4 bits only]

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Bit Operation instruction, Read/Modify/Write type instruction

- **Execution Cycles**
  
  \[1+2a \text{ cycles} \]

- **Instruction Format**
  
  \[
  \begin{array}{c|c|c|c|c|c|c|c|c|c}
  \hline
  & \text{MSB} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \text{u4} & \text{Ri} \\
  \hline
  \end{array}
  \]
7.24 Execution Example

BORL #1,@R3 ; Bit pattern of the instruction: 1001 0000 0001 0011

<table>
<thead>
<tr>
<th>R3</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12345678</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12345679</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N Z V C</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before execution

<table>
<thead>
<tr>
<th>R3</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12345678</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12345679</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N Z V C</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After execution
7.25 BTSTH (Test Higher 4bit of Byte Data in Memory)

Takes the logical AND of the 4-bit immediate data and the higher 4 bits of byte data at memory address Ri places the results in the condition code register (CCR).

- **Assembler Format**
  
  \[
  \text{BTSTH} \quad \#u4, @Ri
  \]

- **Operation**
  
  \[(Ri) & \{u4 << 4\} \quad \text{[Test uses higher 4 bits only]}\]

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the operation result is "1", cleared when the MSB(bit7) is "0".
  
  Z: Set when the operation result is zero, cleared otherwise.
  
  V, C: Unchanged.

- **Classification**

  Bit Operation instruction

- **Execution Cycles**

  2+a cycles

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>[1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad u4 \quad \text{Ri}]</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Execution Example

BTSTH #1, @R3 ; Bit pattern of the instruction: 1000 1001 0001 0011

Before execution

R3

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
</table>

Memory

12345678  12345679
0  1
0  1

NZVC

CCR

0  0  0
0  0  0

After execution

R3

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
</table>

Memory

12345678  12345679
0  1
0  1

NZVC

CCR

0  1  0  0
0  1  0  0
7.26 BTSTL (Test Lower 4bit of Byte Data in Memory)

Takes the logical AND of the 4-bit immediate data and the lower 4 bits of byte data at memory address Ri, places the results in the flag of the condition code register.

- **Assembler Format**
  
  \[
  \text{BTSTL} \ #u4, @Ri
  \]

- **Operation**
  
  \[(Ri) \& u4 \quad \text{[Test uses lower 4 bits only]}\]

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  N: Cleared.
  Z: Set when the operation result is zero, cleared otherwise.
  V, C: Unchanged.

- **Classification**
  
  Bit Operation instruction

- **Execution Cycles**
  
  2+a cycles

- **Instruction Format**
  
  MSB: 1 0 0 0 1 0 0 0  u4
  LSB: Ri
### Execution Example

BTSTL #1, @R3  

; Bit pattern of the instruction: 1000 1000 0011

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>1 2 3 4 5 6 7 8</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12345678</td>
<td>1 0</td>
<td>12345678</td>
</tr>
<tr>
<td>12345679</td>
<td></td>
<td>12345679</td>
</tr>
<tr>
<td>N Z V C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
<td>0 1 0 0</td>
</tr>
</tbody>
</table>

Before execution  After execution
7.27 CALL (Call Subroutine)

This is a branching instruction without a delay slot. After storing the address of the next instruction in the return pointer (RP), branch to the address indicated by label12 relative to the value of the program counter (PC). When calculating the address, double the value of rel11 as a signed extension.

- **Assembler Format**
  
  CALL label12

- **Operation**
  
  PC + 2 → RP
  
  PC + 2 + exts(rel11 × 2) → PC

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Non-delayed branching instruction

- **Execution Cycles**
  
  2 cycles

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rel11</td>
<td></td>
</tr>
</tbody>
</table>
### Execution Example

CALL label ; Bit pattern of the instruction: 1101 0000 1001 0000

...  
label: ; CALL instruction address + 122H

<table>
<thead>
<tr>
<th>PC</th>
<th>RP</th>
<th>PC</th>
<th>RP</th>
</tr>
</thead>
<tbody>
<tr>
<td>F F 8 0 0 0 0 0</td>
<td>x x x x x x x x</td>
<td>F F 8 0 0 1 2 2</td>
<td>F F 8 0 0 0 0 4</td>
</tr>
</tbody>
</table>

Before execution                           After execution
7.28 CALL (Call Subroutine)

This is a branching instruction without a delay slot. After saving the address of the next instruction in the return pointer (RP), a branch to the address indicated by Ri occurs.

- **Assembler Format**
  
  ```
  CALL   @Ri
  ```

- **Operation**
  
  ```
  PC + 2 → RP
  Ri → PC
  ```

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Non-delayed branching instruction

- **Execution Cycles**
  
  2 cycles

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Ri</td>
</tr>
</tbody>
</table>
### Execution Example

CALL @R1 ; Bit pattern of the instruction: 1001 0111 0001 0001

<table>
<thead>
<tr>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>F F F F F 8 0 0</td>
</tr>
<tr>
<td>PC</td>
<td>8 0 0 0 F F F E</td>
</tr>
<tr>
<td>RP</td>
<td>x x x x x x x x</td>
</tr>
</tbody>
</table>
7.29 CALL:D (Call Subroutine)

This is a branching instruction with a delay slot. After saving the address of the next instruction after the delay slot to the return pointer (RP), branch to the address indicated by label12 relative to the value of the program counter (PC). When calculating the address, double the value of rel11 as a signed extension.

- **Assembler Format**
  
  CALL:D label12

- **Operation**
  
  PC + 4 → RP
  
  PC + 2 + exts(rel11 × 2) → PC

- **Flag Change**
  
  N, Z, V, C: Unchanged.

- **Classification**
  
  Delayed branching instruction

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rel11</td>
<td></td>
</tr>
</tbody>
</table>

  MSB (Most Significant Bit) and LSB (Least Significant Bit)
**Execution Example**

CALL: D  label ; Bit pattern of the instruction: 1101 1000 1001 0000

LDI: 8  #0, R2 ; Instruction placed in delay slot

...  

label: ; CALL instruction address + 122H

Before execution of CALL instruction

<table>
<thead>
<tr>
<th>R2</th>
<th>x x x x x x x x</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>FF 8 0 0 0 0 0 0</td>
</tr>
<tr>
<td>RP</td>
<td>x x x x x x x x</td>
</tr>
</tbody>
</table>

After branching

<table>
<thead>
<tr>
<th>R2</th>
<th>0 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>FF 8 0 0 1 2 2</td>
</tr>
<tr>
<td>RP</td>
<td>FF 8 0 0 0 0 4</td>
</tr>
</tbody>
</table>

The instruction placed in delay slot is executed before execution of the branch destination instruction. The value R2 above will vary according to the specifications of the LDI:8 instruction placed in the delay slot.
This is a branching instruction with a delay slot. After saving the address of the next instruction after the delay slot to the return pointer (RP), it branches to the address indicated by Ri.

- **Assembler Format**
  
  
  \[
  \text{CALL:D } @\text{Ri}
  \]

- **Operation**

  
  
  \[
  \begin{align*}
  \text{PC} + 4 & \rightarrow \text{RP} \\
  \text{Ri} & \rightarrow \text{PC}
  \end{align*}
  \]

- **Flag Change**

  
  
  \[
  \begin{array}{cccc}
  \text{N} & \text{Z} & \text{V} & \text{C} \\
  - & - & - & -
  \end{array}
  \]

  
  N, Z, V, C: Unchanged.

- **Classification**

  
  Delayed branching instruction

- **Execution Cycles**

  
  1 cycle

- **Instruction Format**

  
  
  \[
  \begin{array}{cccccccccccc}
  \text{MSB} & & & & & \text{LSB} \\
  1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & \text{Ri}
  \end{array}
  \]
7.30 Execution Example

CHAIN:D @R1 ; Bit pattern of the instruction: 1001 1111 0001 0001
LDI:8 #1, R1 ; Instruction placed in delay slot

| R1 | F F F F F 8 0 0 | R1 | 0 0 0 0 0 0 0 1 |
| PC | 8 0 0 0 F F F E | PC | F F F F 8 0 0 |
| RP | x x x x x x x | RP | 8 0 0 1 0 0 0 2 |

Before execution of CALL instruction  
After branching

The instruction placed in delay slot is executed before execution of the branch destination instruction. The value R2 above will vary according to the specifications of the LDI:8 instruction placed in the delay slot.
7.31 CMP (Compare Immediate Data and Destination Register)

Subtracts the result of the higher 28 bits of 4-bit immediate data with zero extension from the word data in Ri, sets results in the flag of condition code register (CCR).

- Assembler Format
  
  CMP #i4, Ri

- Operation
  
  Ri - extu(i4)

- Flag Change
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V: Set when an overflow has occurred as a result of the operation, cleared otherwise.
  C: Set when a borrow has occurred as a result of the operation, cleared otherwise.

- Classification
  
  Compare instruction, Instruction with delay slot

- Execution Cycles
  
  1 cycle

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i4</td>
<td>Ri</td>
</tr>
</tbody>
</table>
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

7.31

● Execution Example

CMP  #3, R3  ; Bit pattern of the instruction: 1010 1000 0011 0011

R3  0 0 0 0 0 0 0 0 3
    N Z V C

CCR  0 0 0 0

Before execution

R3  0 0 0 0 0 0 0 0 3
    N Z V C

CCR  0 1 0 0

After execution
7.32 CMP (Compare Word Data in Source Register and Destination Register)

Subtracts the word data in Rj from the word data in Ri, sets results in the flag of condition code register (CCR).

- **Assembler Format**
  
  `CMP Rj, Ri`

- **Operation**
  
  `Ri - Rj`

- **Flag Change**
  
  - N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  - Z: Set when the operation result is zero, cleared otherwise.
  - V: Set when an overflow has occurred as a result of the operation, cleared otherwise.
  - C: Set when a borrow has occurred as a result of the operation, cleared otherwise.

- **Classification**
  
  Compare instruction, Instruction with delay slot

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
  MSB
  
  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | Rj | Ri |
  
  LSB

  | C | C | C | C |

  "NZVC"

  "CCCC"
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

7.32

- Execution Example

CMP R2, R3 ; Bit pattern of the instruction: 1010 1010 0010 0011

<table>
<thead>
<tr>
<th></th>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>1 2 3 4 5 6 7 8</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>R3</td>
<td>1 2 3 4 5 6 7 8</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
<td>0 1 0 0</td>
</tr>
</tbody>
</table>

Before execution: NZVC 0000
After execution: NZVC 0100
7.33 CMP2 (Compare Immediate Data and Destination Register)

Subtracts the result of the higher 28 bits of 4-bit immediate (from -16 to -1) data with minus extension from the word data in Ri, sets results in the flag of condition code register (CCR).

- **Assembler Format**
  
  CMP2 #i4, Ri

- **Operation**
  
  Ri - extn(i4)

- **Flag Change**
  
  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V: Set when an overflow has occurred as a result of the operation, cleared otherwise.
  C: Set when a borrow has occurred as a result of the operation, cleared otherwise.

- **Classification**
  
  Compare instruction, Instruction with delay slot

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>i4</td>
<td>Ri</td>
</tr>
</tbody>
</table>
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

7.33

● Execution Example

CMP2 #3, R3 ; Bit pattern of the instruction: 1010 1001 1101 0011

Before execution | After execution

R3: F F F F F F F D | R3: F F F F F F F D
NZVC: 0 0 0 0 | NZVC: 0 1 0 0

CCR: 0 0 0 0 | CCR: 0 1 0 0
This is a step division instruction. This command issued for signed division in which multiplication division register (MDL) contains the dividend and the Ri the divisor, with the quotient stored in the MDL and the remainder in multiplication division register (MDH).

- Assembler Format

```
DIV0S   Ri
```

- Operation

```
MDL[31] → D0
MDL[31] ^ Ri[31] → D1
exts(MDL) → MDH, MDL
```

The word data in MDL is extended to 64 bits, with the higher word in the MDH and the lower word in the MDL. The value of the sign bit in the MDL and Ri is used to set the D0 and D1 flag bits in the system condition code register (SCR).

- Flag Change

```
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

N, Z, V, C: Flags unchanged.

- D1: Set when the divisor and dividend signs are different, cleared when equal.

- D0: Set when the dividend is negative, cleared when positive.

- Classification

Multiply/Divide Instruction

- Execution Cycles

1 cycle
### Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>Ri</td>
</tr>
</tbody>
</table>

### Execution Example

DIV0S R2 ; Bit pattern of the instruction: 1001 0111 0100 0010

Before execution:

- **R2**: 0 F F F F F F F
- **MDH**: 0 0 0 0 0 0 0 0
- **MDL**: F F F F F F 0 0
- **SCR**: x x 0

After execution:

- **R2**: 0 F F F F F F F
- **MDH**: F F F F F F F F
- **MDL**: F F F F F F F 0
- **SCR**: 1 1 0
7.35 DIV0U (Initial Setting Up for Unsigned Division)

This is a step division command. This command issued for unsigned division in which multiplication division register (MDL) contains the dividend and the Ri the divisor, with the quotient stored in the MDL and the remainder in multiplication division register (MDH).

- Assembler Format
  
  \[ \text{DIV0U} \quad \text{Ri} \]

- Operation
  
  \[
  \begin{align*}
  0 & \rightarrow D0 \\
  0 & \rightarrow D1 \\
  0 & \rightarrow \text{MDH}
  \end{align*}
  \]

  The MDH and bits D0 and D1 from system condition code register (SCR) are cleared to "0".

- Flag Change
  
  \[
  \begin{array}{c|c|c|c|c|c|c|c}
  N & Z & V & C & D1 & D0 \\
  \hline
  - & - & - & - & 0 & 0 \\
  \end{array}
  \]

  N, Z, V, C: Flags unchanged.

  D1, D0: Cleared.

- Classification
  
  Multiply/Divide Instruction

- Execution Cycles
  
  1 cycle

- Instruction Format
  
  \[
  \begin{array}{cccccccccccc}
  \text{MSB} & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & \text{Ri} \\
  \text{LSB} & & & & & & & & & & & & \end{array}
  \]
Execution Example

DIV0U R2 ; Bit pattern of the instruction: 1001 0111 0101 0010

<table>
<thead>
<tr>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2 0 F F F F F F F</td>
<td>R2 0 F F F F F F F</td>
</tr>
<tr>
<td>MDH 0 0 0 0 0 0 0 0</td>
<td>MDH 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>MDL 0 F F F F F F F 0</td>
<td>MDL 0 F F F F F F F 0</td>
</tr>
<tr>
<td>D1 D0 T x x 0</td>
<td>D1 D0 T 0 0 0</td>
</tr>
<tr>
<td>SCR</td>
<td>SCR</td>
</tr>
</tbody>
</table>

Before execution | After execution
This is a step division instruction used for unsigned division.

- Assembler Format
  
  DIV1 Ri

- Operation
  
  \{MDH, MDL\} <<= 1  /* 1 bit left shift */
  
  if (D1==1) {
    MDH + Ri \rightarrow temp
  }
  
  else {
    MDH - Ri \rightarrow temp
  }
  
  if ({D0 ^ D1 ^ C} == 0) {
    temp \rightarrow MDH
    1 \rightarrow MDL[0]
  }

- Flag Change
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>C</td>
<td>-</td>
<td>C</td>
</tr>
</tbody>
</table>

  N, V: Unchanged.
  Z: Set when the result of step division is zero, cleared otherwise. Set according to remainder of division results, not according to quotient.
  C: Set when the operation result of step division involves a carry operation, cleared otherwise.

- Classification
  
  Multiply/Divide Instruction

- Execution Cycles
  
  1 cycle
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

7.36

**FR81 Family**

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>Ri</td>
</tr>
</tbody>
</table>

- **Execution Example**

```
DIV1 R2 ; Bit pattern of the instruction: 1001 0111 0110 0010

R2 0 0 F F F F F F F F
MDH 0 0 F F F F F F F F
MDL 0 0 0 0 0 0 0 0 0 0
D1 D0 T 0 0 0 0
SCR 0 0 0

Before execution

R2 0 0 F F F F F F F F
MDH 0 1 0 0 0 0 0 0
MDL 0 0 0 0 0 0 0 1
D1 D0 T 0 0 0 0
SCR 0 0 0

NZVC 0 0 0 0
CCR 0 0 0 0

After execution
```
This is a step division instruction used for signed division.

- **Assembler Format**
  
  ```
  DIV2 Ri
  ```

- **Operation**
  ```
  if (D1==1) {
      MDH + Ri → temp
  } else {
      MDH - Ri → temp
  }
  if (Z==1) {
      0 → MDH
  }
  ```

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>C</td>
<td>-</td>
<td>C</td>
</tr>
</tbody>
</table>

  N, V: Unchanged.
  Z: Set when the result of step division is zero, cleared otherwise. Set according to remainder of division results, not according to quotient.
  C: Set when the operation result of step division involves a carry or borrow operation, cleared otherwise.

- **Classification**
  
  Multiply/Divide Instruction

- **Execution Cycles**
  
  c cycle
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

7.37

FR81 Family

- Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Ri</td>
</tr>
</tbody>
</table>

- Execution Example

DIV2  R2 ; Bit pattern of the instruction: 1001 0111 0111 0010

Before execution After execution

\[
\begin{array}{c|c|c}
R2 & 0 & 0 \\
\hline
MDH & 0 & 0 \\
MDL & 0 & 0 \\
\hline
SCR & 0 & 0 \\
NZVC & 0 & 0 \\
\hline
OCR & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
R2 & 0 & 0 \\
\hline
MDH & 0 & 0 \\
MDL & 0 & 0 \\
\hline
SCR & 0 & 0 \\
NZVC & 0 & 0 \\
\hline
OCR & 0 & 0 \\
\end{array}
\]

178FUJITSU MICROELECTRONICS LIMITED CM71-00105-1E
7.38 DIV3 (Correction When Remain is zero)

This is a step division instruction used for signed division.

- **Assembler Format**
  
  DIV3

- **Operation**
  
  if (Z==1) {
    MDL + 1 → MDL
  }

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Multiply/Divide Instruction

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th></th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>1</td>
<td>1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
7.38 Execution Example

DIV3 ; Bit pattern of the instruction: 1001 1111 0110 0000

R2

<table>
<thead>
<tr>
<th>MDH</th>
<th>MDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>0000 000F</td>
</tr>
</tbody>
</table>

D1 D0 T

SCR

<table>
<thead>
<tr>
<th>N Z V C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
</tr>
</tbody>
</table>

CCR

Before execution

R2

<table>
<thead>
<tr>
<th>MDH</th>
<th>MDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>0000 0010</td>
</tr>
</tbody>
</table>

D1 D0 T

SCR

<table>
<thead>
<tr>
<th>N Z V C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
</tr>
</tbody>
</table>

CCR

After execution
This is a step division instruction used for signed division.

- **Assembler Format**
  
  DIV4S

- **Operation**
  
  if (D1==1) {
    0 - MDL → MDL.
  }

- **Flag Change**
  
  N, Z, V, C: Unchanged.

- **Classification**
  
  Multiply/Divide Instruction

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Execution Example

DIV4S ; Bit pattern of the instruction: 1001 1111 0111 0000

Before execution

R2 0 0 F F F F F F F F
MDH 0 0 0 0 0 0 0 0 0 0
MDL 0 0 0 0 0 0 0 F

Scr 1 1 0

N Z V C 0 0 0 0

Ccr 0 0 0 0

After execution

R2 0 0 F F F F F F F F
MDH 0 0 0 0 0 0 0 0 0 0
MDL F F F F F F F 1

Scr 1 1 0

N Z V C 0 0 0 0

Ccr 0 0 0 0
7.40  DMOV (Move Word Data from Direct Address to Register)

Transfers, to R13 the word data at the direct address corresponding to 4 times the value of dir8. The value of dir8 \( \times 4 \) is specified as dir10.

- **Assembler Format**
  
  
  DMOV @dir10, R13

- **Operation**
  
  
  \((\text{dir8} \times 4) \rightarrow R13\)

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**

  Direct Addressing Instructions, Instruction with delay slot

- **Execution Cycles**

  b cycle

- **Instruction Format**

  \[
  \begin{array}{cccccccc}
  \text{MSB} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
  \text{LSB} & \multicolumn{1}{|c}{\text{dir8}} \\
  \end{array}
  \]
Execution Example

DMOV @88H, R13  ; Bit pattern of the instruction: 0000 1000 0010 0010

Before execution

After execution
Transfers word data in R13 to the direct address corresponding to 4 times the value of dir8. The value of dir8 × 4 is specified as dir10.

- **Assembler Format**
  
  DMOV R13,dir10

- **Operation**
  
  R13 → (dir8 × 4)

- **Flag Change**
  
  N, Z, V, C: Unchanged.

- **Classification**
  
  Direct Addressing Instructions, Instruction with delay slot

- **Execution Cycles**
  
  a cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>dir8</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 7 DETAILED EXECUTION INSTRUCTIONS

7.41

Execution Example

DMOV R13,@54H ; Bit pattern of the instruction: 0001 1000 0001 0101

R13  |  8 9 A B C D E F  |
---|------------------|
R13  |  8 9 A B C D E F  |

Memory

50H  |  x x x x x x x x  |
54H  |  x x x x x x x x  |
58H  |  x x x x x x x x  |

Before execution

50H  |  x x x x x x x x  |
54H  |  8 9 A B C D E F  |
58H  |  x x x x x x x x  |

After execution
7.42 DMOV (Move Word Data from Direct Address to Post Increment Register Indirect Address)

Transfers the word data at the direct address corresponding to 4 times the value of dir8 to the address indicated in R13. After the data transfer, it increments the value of R13 by 4. The value of dir8 × 4 is specified as dir10.

- Assembler Format
  
  DMOV @dir10, @R13+

- Operation
  
  \[(\text{dir8} \times 4) \rightarrow (\text{R13})\]
  
  \[\text{R13 + 4} \rightarrow \text{R13}\]

- Flag Change
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification
  
  Direct Addressing Instructions

- Execution Cycles
  
  1+2a cycles

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

  dir8
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

7.42

Execution Example

DMOV  @88H,@R13+  ; Bit pattern of the instruction: 0000 1100 0010 0010

Before execution

<table>
<thead>
<tr>
<th>R13</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000088</td>
<td>1 4 1 4 2 1 3 5</td>
</tr>
<tr>
<td>FFFF1248</td>
<td>x x x x x x x</td>
</tr>
<tr>
<td>FFFF124C</td>
<td>x x x x x x x</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R13</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000088</td>
<td>1 4 1 4 2 1 3 5</td>
</tr>
<tr>
<td>FFFF1248</td>
<td>1 4 1 4 2 1 3 5</td>
</tr>
<tr>
<td>FFFF124C</td>
<td>x x x x x x x</td>
</tr>
</tbody>
</table>
7.43 DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address)

Transfers the word data at the address indicated in R13 to the direct address corresponding to 4 times the value dir8. After the data transfer, it increments the value of R13 by 4. The value of dir8 × 4 is specified as dir10.

- **Assembler Format**
  
  DMOV @R13+,@dir10

- **Operation**

  (R13) → (dir8 × 4)
  
  R13 + 4 → R13

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**

  Direct Addressing Instructions

- **Execution Cycles**

  1+2a cycles

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>dir8</td>
<td></td>
</tr>
</tbody>
</table>
### Execution Example

**DMOV @R13+,@54H** : Bit pattern of the instruction: 0001 1100 0001 0101

<table>
<thead>
<tr>
<th>R13</th>
<th>F F F F 1 2 4 8</th>
<th>R13</th>
<th>F F F F 1 2 4 C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td></td>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>00000054</td>
<td>x x x x x x x</td>
<td>00000054</td>
<td>8 9 4 7 9 1 A F</td>
</tr>
<tr>
<td>FFFF1248</td>
<td>8 9 4 7 9 1 A F</td>
<td>FFFF1248</td>
<td>8 9 4 7 9 1 A F</td>
</tr>
<tr>
<td>FFFF124C</td>
<td>x x x x x x x</td>
<td>FFFF124C</td>
<td>x x x x x x x</td>
</tr>
</tbody>
</table>

Before execution       After execution
7.44 DMOV (Move Word Data from Direct Address to Pre Decrement Register Indirect Address)

Decrements the value of R15 by 4, then transfers the word data at the direct address corresponding to 4 times the value of dir8 to the address indicated in R15. The value of dir8 × 4 is specified as dir10.

- **Assembler Format**
  
  ```
  DMOV @dir10, @-R15
  ```

- **Operation**
  
  ```
  R15 - 4 → R15
  (dir8 × 4) → (R15)
  ```

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Direct Addressing Instructions

- **Execution Cycles**
  
  1+2a cycles

- **Instruction Format**
  
  MSB
  
  | | | | 1 | 0 | 0 | 1 | dir8 |
  | | | | | | | |   |
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS
7.44

● Execution Example

DMOV   @2CH,@-R15 ; Bit pattern of the instruction: 0000 1011 0000 1011

R15

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>8</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000002C</td>
<td>8</td>
<td>2</td>
<td>A</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>A</td>
</tr>
</tbody>
</table>

Memory

7FFFFFF84

x

x

x

x

x

7FFFFFF88

x

x

x

x

x

Before execution

R15

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>8</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000002C</td>
<td>8</td>
<td>2</td>
<td>A</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>A</td>
</tr>
</tbody>
</table>

Memory

7FFFFFF84

x

x

x

x

x

7FFFFFF88

x

x

x

x

x

After execution
7.45 DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address)

Transfers the word data at the address indicated in R15 to the direct address corresponding to 4 times the value dir8. After the data transfer, it increments the value of R15 by 4. The value of dir8 × 4 is specified as dir10.

- Assembler Format
  
  DMOV @R15+,@dir10

- Operation
  
  (R15) → (dir8 × 4)
  R15 + 4 → R15

- Flag Change
  
  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  - & - & - & - \\
  \end{array}
  \]

  N, Z, V, C: Unchanged.

- Classification
  
  Direct Addressing Instructions

- Execution Cycles
  
  1+2a cycles

- Instruction Format
  
  \[
  \begin{array}{cccccccc}
  \text{MSB} & \\
  0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
  \end{array}
  \]

  dir8
## Execution Example

DMOV @R15+,@38H ; Bit pattern of the instruction: 0001 1011 0000 1110

<table>
<thead>
<tr>
<th>R15</th>
<th>7 F F E E 8 0</th>
<th>R15</th>
<th>7 F F E E 8 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000038</td>
<td>x x x x x  x x x</td>
<td>00000038</td>
<td>8 3 4 3 8 3 4 A</td>
</tr>
<tr>
<td>7FFEEE80</td>
<td>8 3 4 3 8 3 4 A</td>
<td>7FFEEE80</td>
<td>8 3 4 3 8 3 4 A</td>
</tr>
<tr>
<td>7FFEEE84</td>
<td>x x x x x x x</td>
<td>7FFEEE84</td>
<td>x x x x x x x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000038</td>
<td>00000038</td>
</tr>
<tr>
<td>7FFEEE80</td>
<td>7FFEEE80</td>
</tr>
<tr>
<td>7FFEEE84</td>
<td>7FFEEE84</td>
</tr>
</tbody>
</table>

Bit pattern of the instruction: 0001 1011 0000 1110
7.46  DMOVB (Move Byte Data from Direct Address to Register)

Transfers the byte data at the address indicated by the value dir8 to R13. Uses zeros to extend the higher 24 bits of data.

- Assembler Format
  
  DMOVB  @dir8, R13

- Operation
  
  (dir8) → R13

- Flag Change
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification
  
  Direct Addressing Instructions, Instruction with delay slot

- Execution Cycles
  
  b cycle

- Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

  dir8
 Execution Example

DMOVB @91H, R13 ; Bit pattern of the instruction: 0000 1010 1001 0001

Before execution

R13: x x x x x x x x

Memory

90  x x
91  3 2
92  x x

After execution

R13: 0 0 0 0 0 0 3 2

Memory

90  x x
91  3 2
92  x x
7.47 DMOVB (Move Byte Data from Register to Direct Address)

Transfers the byte data from R13 to the direct address indicated by the value dir8.

- Assembler Format
  
  DMOVB R13, @dir8

- Operation
  
  R13 → (dir8)

- Flag Change
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification
  
  Direct Addressing Instructions, Instruction with delay slot

- Execution Cycles
  
  a cycle

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>1 1 0 1 0 1 0</th>
<th>dir8</th>
<th>LSB</th>
</tr>
</thead>
</table>
### Execution Example

DMOVB R13, @53H ; Bit pattern of the instruction: 0001 1010 0101 0011

<table>
<thead>
<tr>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R13 F F F F F F F E</td>
<td>R13 F F F F F F F E</td>
</tr>
<tr>
<td>52</td>
<td>x x</td>
</tr>
<tr>
<td>53</td>
<td>x x</td>
</tr>
<tr>
<td>54</td>
<td>x x</td>
</tr>
</tbody>
</table>
7.48 DMOVB (Move Byte Data from Direct Address to Post Increment Register Indirect Address)

Moves the byte data at the direct address indicated by the value dir8 to the address indicated by R13. After the data transfer, it increments the value of R13 by 1.

- **Assembler Format**
  
  DMOVB @dir8,@R13+

- **Operation**
  
  (dir8) → (R13)
  
  R13 + 1 → R13

- **Flag Change**
  
  N, Z, V, C: Unchanged.

- **Classification**
  
  Direct Addressing Instructions

- **Execution Cycles**
  
  1+2a cycles

- **Instruction Format**
  
  MSB |          |          |          |          |          |          |          |          | dir8  
  | 0    | 0        | 0        | 0        | 1        | 1        | 1        | 0        |       |
Execution Example

DMOVB @71H,@R13+ ; Bit pattern of the instruction: 0000 1110 0111 0001
7.49 DMOVB (Move Byte Data from Post Increment Register Indirect Address to Direct Address)

Transfers the byte data at the address indicated by R13 to the direct address indicated by the value dir8. After the data transfer, it increments the value of R13 by 1.

- **Assembler Format**
  
  DMOVB @R13+, @dir8

- **Operation**

  \[(R13) \rightarrow (\text{dir8})\]
  
  \[\text{R13} + 1 \rightarrow \text{R13}\]

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**

  Direct Addressing Instructions

- **Execution Cycles**

  1+2a cycles

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>dir8</td>
</tr>
</tbody>
</table>
Execution Example

DMOVB @R13+,@57H ; Bit pattern of the instruction: 0001 1110 0101 0111

Before execution

R13

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000057</td>
</tr>
<tr>
<td>FF801220</td>
</tr>
<tr>
<td>FF801221</td>
</tr>
</tbody>
</table>

After execution

R13

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000057</td>
</tr>
<tr>
<td>FF801220</td>
</tr>
<tr>
<td>FF801221</td>
</tr>
</tbody>
</table>
7.50 DMOVH (Move Halfword Data from Direct Address to Register)

Transfers the half-word data at the direct address corresponding to 2 times the value dir8 to R13. Uses zeros to extend the higher 16 bits of data. The value of dir8 × 2 is specified as dir9.

- **Assembler Format**
  
  DMOVH @dir9, R13

- **Operation**
  
  (dir8 × 2) → R13

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Direct Addressing Instructions, Instruction with delay slot

- **Execution Cycles**
  
  b cycle

- **Instruction Format**
  
  MSB
  
  | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | dir8 |
  
  LSB
7.50

Execution Example

DMOVH @88H, R13 ; Bit pattern of the instruction: 0000 1001 0100 0100

Before execution

R13

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>86</td>
</tr>
<tr>
<td>88</td>
</tr>
<tr>
<td>8A</td>
</tr>
</tbody>
</table>

After execution

R13

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>86</td>
</tr>
<tr>
<td>88</td>
</tr>
<tr>
<td>8A</td>
</tr>
</tbody>
</table>
7.51 DMOVH (Move Halfword Data from Register to Direct Address)

Transfers the half-word data from R13 to the direct address corresponding to 2 times the value dir8. The value of dir8 × 2 is specified as dir9.

- **Assembler Format**
  
  DMOVH R13, @dir9

- **Operation**
  
  R13 → (dir8 × 2)

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**

  Direct Addressing Instructions, Instruction with delay slot

- **Execution Cycles**

  a cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

  dir8
 Execution Example

DMOVH R13, @52H ; Bit pattern of the instruction: 0001 1001 0010 1001

Before execution

<table>
<thead>
<tr>
<th>Memory</th>
<th>50</th>
<th>52</th>
<th>54</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x</td>
<td>x x x x</td>
<td>x x x x</td>
<td></td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>Memory</th>
<th>50</th>
<th>52</th>
<th>54</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x</td>
<td>A E 8 6</td>
<td>x x x x</td>
<td></td>
</tr>
</tbody>
</table>
7.52 DMOVH (Move Halfword Data from Direct Address to Post Increment Register Indirect Address)

Transfers the half-word data at the direct address corresponding to 2 times the value dir8 to the address indicated by R13. After the data transfer, it increments the value of R13 by 2. The value of dir8 × 2 is specified as dir9.

- Assembler Format
  
  DMOVH @dir9,@R13+

- Operation
  
  (dir8 × 2) → (R13)
  R13 + 2 → R13

- Flag Change
  
  N, Z, V, C: Unchanged.

- Classification
  
  Direct Addressing Instructions

- Execution Cycles
  
  1+2a cycles

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>dir8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 7 DETAILED EXECUTION INSTRUCTIONS
7.52

**FR81 Family**

- **Execution Example**

  DMOVH @88H, @R13+ ; Bit pattern of the instruction: 0000 1101 0100 0100

<table>
<thead>
<tr>
<th>R13</th>
<th>F F 0 0 0 0 5 2</th>
<th>R13</th>
<th>F F 0 0 0 5 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>1 3 7 4</td>
<td>Memory</td>
<td>1 3 7 4</td>
</tr>
<tr>
<td>00000088</td>
<td>1 3 7 4</td>
<td>00000088</td>
<td>1 3 7 4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>FF000052</td>
<td>x x x x</td>
<td>FF000052</td>
<td>1 3 7 4</td>
</tr>
<tr>
<td>FF000054</td>
<td>x x x x</td>
<td>FF000054</td>
<td>x x x x</td>
</tr>
</tbody>
</table>

Before execution | After execution
TRANSfers the half-word data at the address indicated by R13 to the direct address corresponding to 2 times the value dir8. After the data transfer, it increments the value of R13 by 2. The value of dir8 × 2 is specified as dir9.

- Assembler Format
  DMOVH @R13+, @dir9

- Operation
  (R13) → (dir8 × 2)
  R13 + 2 → R13

- Flag Change
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification
  Direct Addressing Instructions

- Execution Cycles
  1 + 2a cycles

- Instruction Format


Execution Example

DMOVH @R13+,@52H ; Bit pattern of the instruction: 0001 1101 0010 1001

Before execution

<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000052</td>
<td>8 9 3 3</td>
</tr>
<tr>
<td>FF801220</td>
<td>8 9 3 3</td>
</tr>
<tr>
<td>FF801222</td>
<td>x x x x</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000052</td>
<td>8 9 3 3</td>
</tr>
<tr>
<td>FF801220</td>
<td>8 9 3 3</td>
</tr>
<tr>
<td>FF801222</td>
<td>x x x x</td>
</tr>
</tbody>
</table>
7.54 ENTER (Enter Function)

This instruction is used for stack frame generation processing for high level languages. The value \( u_8 \) is calculated as an unsigned value. The value of \( u_8 \times 4 \) is specified as \( u_{10} \).

- **Assembler Format**

  \[
  \text{ENTER} \ #u_{10}
  \]

- **Operation**

  \[
  \begin{align*}
  R14 &\rightarrow (R15-4) \\
  R15 - 4 &\rightarrow R14 \\
  R15 - \text{extu}(u_8 \times 4) &\rightarrow R15
  \end{align*}
  \]

- **Flag Change**

  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  - & - & - & - \\
  \end{array}
  \]

  N, Z, V, C: Unchanged.

- **Classification**

  Other instructions

- **Execution Cycles**

  \( 1+a \) cycles

- **Instruction Format**

  \[
  \begin{array}{cccccccc}
  \text{MSB} & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
  \text{LSB} & \text{u}_8 \\
  \end{array}
  \]
 Execution Example

ENTER  #0CH ; Bit pattern of the instruction: 0000 1111 0000 0011

<table>
<thead>
<tr>
<th>R14</th>
<th>8 0 0 0 0 0 0 0</th>
<th>R14</th>
<th>7 F F F F F 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R15</td>
<td>7 F F F F F 8</td>
<td>R15</td>
<td>7 F F F F E C</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>7FFFFFEC</th>
<th>x x x x x x x</th>
<th>7FFFFFEC</th>
<th>x x x x x x x</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FFFFF0</td>
<td>x x x x x x x</td>
<td>7FFFFF0</td>
<td>x x x x x x x</td>
</tr>
<tr>
<td>7FFFFF4</td>
<td>x x x x x x x</td>
<td>7FFFFF4</td>
<td>8 0 0 0 0 0 0</td>
</tr>
<tr>
<td>7FFFFF8</td>
<td>x x x x x x x</td>
<td>7FFFFF8</td>
<td>x x x x x x x</td>
</tr>
<tr>
<td>7FFFFFFC</td>
<td>x x x x x x x</td>
<td>7FFFFFFC</td>
<td>x x x x x x x</td>
</tr>
<tr>
<td>80000000</td>
<td>x x x x x x x</td>
<td>80000000</td>
<td>x x x x x x x</td>
</tr>
</tbody>
</table>

Before execution   After execution
7.55 EOR (Exclusive Or Word Data of Source Register to Data in Memory)

Takes the logical exclusive OR of the word data at memory address Ri and the word data in Rj, stores the results to the memory address corresponding to Ri.

- **Assembler Format**
  
  EOR  Rj,@Ri

- **Operation**
  
  (Ri) ^ Rj → (Ri)

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  - N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  - Z: Set when the operation result is zero, cleared otherwise.
  - V, C: Unchanged.

- **Classification**
  
  Logical Calculation instruction, Read/Modify/Write type instruction

- **Execution Cycles**
  
  1+2a cycles

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>10011100</td>
<td>Rj</td>
</tr>
<tr>
<td></td>
<td>Ri</td>
</tr>
</tbody>
</table>
Execution Example

EOR R2, @R3  ; Bit pattern of the instruction: 1001 1100 0010 0011

Before execution After execution
7.56  EOR (Exclusive Or Word Data of Source Register to Destination Register)

Takes the logical exclusive OR of the word data in Ri and the word data in Rj, stores the results to Ri.

- **Assembler Format**
  
  EOR   Rj, Ri

- **Operation**
  
  Ri ^ Rj -> Ri

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V, C: Unchanged.

- **Classification**
  
  Logical Calculation instruction, Instruction with delay slot

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | Rj | Ri |

  MSB  |  |  |  |  |  |  |  |  |  |
Execution Example

EOR R2, R3 ; Bit pattern of the instruction: 1001 1010 0010 0011

Before execution

\[ \begin{array}{c|c|c|c|c|c|c} 
R2 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
R3 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\hline 
N & Z & V & C & & & & & \\
CCR & 0 & 0 & 0 & 0 & & & & \\
\end{array} \]

After execution

\[ \begin{array}{c|c|c|c|c|c|c} 
R2 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
R3 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
\hline 
N & Z & V & C & & & & & \\
CCR & 0 & 0 & 0 & 0 & & & & \\
\end{array} \]
7.57 EORB (Exclusive Or Byte Data of Source Register to Data in Memory)

Takes the logical exclusive OR of the byte data at memory address Ri and the byte data in Rj, stores the results to the memory address corresponding to Ri.

- **Assembler Format**
  
  EORB   Rj,@Ri

- **Operation**
  
  \[(Ri) \oplus Rj \rightarrow (Ri)\]

- **Flag Change**
  
  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  C & C & - & - \\
  \end{array}
  \]
  
  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  
  Z: Set when the operation result is zero, cleared otherwise.
  
  V, C: Unchanged.

- **Classification**
  
  Logical Calculation instruction, Read/Modify/Write type instruction

- **Execution Cycles**
  
  1+2a cycles

- **Instruction Format**
  
  \[
  \begin{array}{cccccccc}
  \text{MSB} & 1 & 0 & 0 & 1 & 1 & 1 & 0 & \text{Rj} & \text{Ri} \\
  \text{LSB} & & & & & & & & & \\
  \end{array}
  \]
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS
7.57

**Execution Example**

EORB  R2,@R3 ; Bit pattern of the instruction: 1001 1110 0010 0011

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0011</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>1 0</td>
</tr>
<tr>
<td>12345679</td>
<td>0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N Z V C</th>
<th>N Z V C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

Before execution          After execution
7.58 EORH (Exclusive Or Halfword Data of Source Register to Data in Memory)

Takes the logical exclusive OR of the half-word data at memory address Ri and the half-word data in Rj, stores the results to the memory address corresponding to Ri.

- **Assembler Format**
  
  EORH Rj,@Ri

- **Operation**
  
  \((Ri) \oplus Rj \rightarrow (Ri)\)

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N: Set when the MSB(bit15) of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V, C: Unchanged.

- **Classification**
  
  Logical Calculation instruction, Read/Modify/Write type instruction

- **Execution Cycles**
  
  \(1+2a\) cycles

- **Instruction Format**
  
  MSB
  
  \[ \begin{array}{cccccccc}
  1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
  Rj & & & & & & & Ri
  \end{array} \]
Execution Example

EORH R2, @R3 ; Bit pattern of the instruction: 1001 1101 0010 0011

Before execution

<table>
<thead>
<tr>
<th>R2</th>
<th>0 0 0 0 1 1 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
</tr>
<tr>
<td>12345678</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>1234567A</td>
<td>N Z V C</td>
</tr>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R2</th>
<th>0 0 0 0 1 1 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
</tr>
<tr>
<td>12345678</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>1234567A</td>
<td>N Z V C</td>
</tr>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>
7.59 EXTSB (Sign Extend from Byte Data to Word Data)

Extends the byte data indicated by Ri to word data as signed binary value.

- Assembler Format
  
  \texttt{EXTSB \ Ri}

- Operation
  
  \texttt{exts(Ri}[7:0]\texttt{) \rightarrow Ri \ [byte \rightarrow word]}

- Flag Change
  
  \begin{array}{cccc}
  \text{N} & \text{Z} & \text{V} & \text{C} \\
  \text{--} & \text{--} & \text{--} & \text{--} \\
  \end{array}

  \text{N, Z, V, C: Unchanged.}

- Classification
  
  Other instructions, Instruction with delay slot

- Execution Cycles
  
  1 cycle

- Instruction Format
  
  \begin{array}{cccccccc}
  \text{MSB} & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & \text{Ri} \\
  \text{LSB} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} & \text{\ldots} \end{array}
Execution Example

 EXTSB  R1 ; Bit pattern of the instruction: 1001 0111 1000 0001

Before execution

R1 0 0 0 0 0 A B

After execution

R1 F F F F F F A B
7.60 EXTSH (Sign Extend from Byte Data to Word Data)

Extends the half-word data indicated by Ri to word data as a signed binary value.

- **Assembler Format**
  
  ```
  EXTSH Ri
  ```

- **Operation**
  
  ```
  exts(Ri[15:0]) → Ri  [half-word → word]
  ```

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**

  Other instructions, Instruction with delay slot

- **Execution Cycles**

  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Ri</td>
<td></td>
</tr>
</tbody>
</table>

  ```
Execution Example

EXTSH R1 ; Bit pattern of the instruction: 1001 0111 1010 0001

Before execution

After execution
EXTUB (Unsign Extend from Byte Data to Word Data)

Extends the byte data indicated by Ri to word data as an unsigned binary value.

- **Assembler Format**
  
  EXTUB Ri

- **Operation**
  
  extu(Ri[7:0]) → Ri [byte → word]

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**

  Other instructions, Instruction with delay slot

- **Execution Cycles**

  1 cycle

- **Instruction Format**

  MSB  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | Ri  
  LSB  |    |    |    |    |    |    |    |    |    |    |     

  ```
  1 0 0 1 0 1 1 1 1 0 0 1
  ```
7.61 Execution Example

EXTUB R1 ; Bit pattern of the instruction: 1001 0111 1001 0001

Before execution

R1

F F F F F F F

After execution

R1

0 0 0 0 0 0 F F
Extends the half-word data indicated by Ri to word data as an unsigned binary value.

- **Assembler Format**
  
  \[ \text{EXTUH} \; \text{Ri} \]

- **Operation**
  
  \[ \text{extu}(\text{Ri}[15:0]) \rightarrow \text{Ri} \; \text{[half-word} \rightarrow \text{word]} \]

- **Flag Change**
  
  \[
  \begin{array}{cccc}
  \text{N} & \text{Z} & \text{V} & \text{C} \\
  - & - & - & -
  \end{array}
  \]

  N, Z, V, C: Unchanged.

- **Classification**
  
  Other instructions, Instruction with delay slot

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
  MSB

  \[
  \begin{array}{cccccccccccc}
  1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & \text{Ri}
  \end{array}
  \]
Execution Example

`EXTUH R1 ; Bit pattern of the instruction: 1001 0111 1011 0001`

Before execution

<table>
<thead>
<tr>
<th>R1</th>
<th>F F F F F F F F</th>
</tr>
</thead>
</table>

After execution

<table>
<thead>
<tr>
<th>R1</th>
<th>0 0 0 0 F F F F</th>
</tr>
</thead>
</table>
FR81 Family

7.63 FABSs (Single Precision Floating Point Absolute Value)

Loads the absolute value FRj to FRi.

- **Assembler Format**
  
  FABSs FRj, FRi

- **Operation**
  
  \[ |FRj| \rightarrow FRi \]

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0 0 0 0 0 1 1 1 1 0 1 0 1 1 0 0</td>
</tr>
<tr>
<td>(n+2)</td>
<td>- - FRj -</td>
</tr>
</tbody>
</table>

- **EIT Occurrence and Detection**
  
  An invalid instruction exception (FPU absence error), an interrupt is detected.
7.64  FADDs (Single Precision Floating Point Add)

FRk is added to FRj, and its result is stored in FRi.

- Assembler Format
  
  FADDs   FRk, FRj, FRi

- Operation
  
  FRk + FRj → FRi

- Classification
  
  Single-precision floating point instruction, FR81 family

- Execution Cycles
  
  1 cycle

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0 0 0 0 0 1 1 1 1 0 1 0 0 0 0</td>
</tr>
<tr>
<td>(n+2)</td>
<td>-   FRk FRj FRi</td>
</tr>
</tbody>
</table>

- EIT Occurrence and Detection
  
  An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.
Calculation result and exception flag

<table>
<thead>
<tr>
<th></th>
<th>+0</th>
<th>-0</th>
<th>+Norm</th>
<th>-Norm</th>
<th>+INF</th>
<th>-INF</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>+0/(-)</td>
<td>-0/(-)</td>
<td>+Norm/(X)</td>
<td>-Norm/(X)</td>
<td>+INF/(-)</td>
<td>-INF/(-)</td>
<td>QNaN/(-)</td>
<td>QNaN/V</td>
</tr>
<tr>
<td>-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Norm</td>
<td>+Norm/(X)</td>
<td>*1</td>
<td>*2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Norm</td>
<td>-Norm/(X)</td>
<td>*2</td>
<td>*1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+INF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-INF</td>
<td>-INF/(-)</td>
<td></td>
<td></td>
<td></td>
<td>QNaN/V</td>
<td>-INF/(-)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN/V</td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*1: +INF/0,X or -INF/P.X or ± Norm/(X)
*2: ± 0/(-) or ± 0/U or ± Norm/(X)
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

7.65  FBcc (Floating Point Conditional Branch)

This is a branching instruction without a delay slot. If conditions specified for each instruction are satisfied, control branches to the address indicated by label17 relative to the program counter (PC). The rel16 value is doubled and its sign is extended. If conditions are not satisfied, no branching occurs.

- **Assembler Format**
  
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>cc</th>
<th>Contents</th>
<th>conditions (FCC field)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBN</td>
<td>0000</td>
<td>Branch Never</td>
<td>Always unsatisfied</td>
</tr>
<tr>
<td>FBA</td>
<td>1111</td>
<td>Branch Always</td>
<td>Always satisfied</td>
</tr>
<tr>
<td>FBNE</td>
<td>0111</td>
<td>Branch Not Equal</td>
<td>L or G or U</td>
</tr>
<tr>
<td>FBE</td>
<td>1000</td>
<td>Branch Equal</td>
<td>E</td>
</tr>
<tr>
<td>FBLG</td>
<td>0110</td>
<td>Branch Less or Greater</td>
<td>L or G</td>
</tr>
<tr>
<td>FBUE</td>
<td>1001</td>
<td>Branch Unordered or Equal</td>
<td>E or U</td>
</tr>
<tr>
<td>FBUL</td>
<td>0101</td>
<td>Branch Unordered or Less</td>
<td>L or U</td>
</tr>
</tbody>
</table>

  The FBN instruction has no operand.

- **Operation**
  
  ```
  if (condition) {
      PC + 4 + exts(rel16 × 2) → PC
  }
  ```

  The branching conditions of each instruction are shown in Table 7.65-1.
Table 7.65-1 Branching conditions of FBcc instruction (2 / 2)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>cc</th>
<th>Contents</th>
<th>conditions (FCC field)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBGE</td>
<td>1010</td>
<td>Branch Greater or Equal</td>
<td>G or E</td>
</tr>
<tr>
<td>FBL</td>
<td>0100</td>
<td>Branch Less</td>
<td>L</td>
</tr>
<tr>
<td>FBUGE</td>
<td>1011</td>
<td>Branch Unordered or Greater or Equal</td>
<td>U or G or E</td>
</tr>
<tr>
<td>FBUG</td>
<td>0011</td>
<td>Branch Unorder or Greater</td>
<td>G or U</td>
</tr>
<tr>
<td>FBLE</td>
<td>1100</td>
<td>Branch Less or Equal</td>
<td>L or E</td>
</tr>
<tr>
<td>FBG</td>
<td>0010</td>
<td>Branch Greater</td>
<td>G</td>
</tr>
<tr>
<td>FBULE</td>
<td>1101</td>
<td>Branch Unordered or Less or Equal</td>
<td>E or L or U</td>
</tr>
<tr>
<td>FBU</td>
<td>0001</td>
<td>Branch Unordered</td>
<td>U</td>
</tr>
<tr>
<td>FBO</td>
<td>1110</td>
<td>Branch Ordered</td>
<td>E or L or G</td>
</tr>
</tbody>
</table>

**Classification**
Floating point instruction, FR81 family

**Execution Cycles**
2 cycles

**Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**EIT Occurrence and Detection**
An interrupt is detected.
This is a branching instruction having a delay slot. If conditions specified for each instruction are satisfied, control branches to the address indicated by label17 relative to the program counter (PC). The rel16 value is doubled and its sign is extended. If conditions are not satisfied, no branching occurs.

### Assembler Format

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Label17 Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBN:D</td>
<td>0000</td>
</tr>
<tr>
<td>FBA:D</td>
<td>1111</td>
</tr>
<tr>
<td>FBNE:D</td>
<td>0111</td>
</tr>
<tr>
<td>FBE:D</td>
<td>1000</td>
</tr>
<tr>
<td>FBLG:D</td>
<td>0110</td>
</tr>
<tr>
<td>FBUE:D</td>
<td>1001</td>
</tr>
<tr>
<td>FBUL:D</td>
<td>0101</td>
</tr>
</tbody>
</table>

The FBN:D instruction has no operand.

### Operation

```
if (condition) {
    PC + 4 + exts(rel16 \times 2) \rightarrow PC
}
```

The branching conditions of each instruction are shown in Table 7.66-1.

### Table 7.66-1 Branching conditions of FBcc:D instruction (1 / 2)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>cc</th>
<th>Contents</th>
<th>conditions (FCC field)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBN:D</td>
<td>0000</td>
<td>Branch Never</td>
<td>Always unsatisfied</td>
</tr>
<tr>
<td>FBA:D</td>
<td>1111</td>
<td>Branch Always</td>
<td>Always satisfied</td>
</tr>
<tr>
<td>FBNE:D</td>
<td>0111</td>
<td>Branch Not Equal</td>
<td>L or G or U</td>
</tr>
<tr>
<td>FBE:D</td>
<td>1000</td>
<td>Branch Equal</td>
<td>E</td>
</tr>
<tr>
<td>FBLG:D</td>
<td>0110</td>
<td>Branch Less or Greater</td>
<td>L or G</td>
</tr>
<tr>
<td>FBUE:D</td>
<td>1001</td>
<td>Branch Unordered or Equal</td>
<td>E or U</td>
</tr>
<tr>
<td>FBUL:D</td>
<td>0101</td>
<td>Branch Unordered or Less</td>
<td>L or U</td>
</tr>
</tbody>
</table>
Classification
Floating point instruction, FR81 family

Execution Cycles
1 cycle

Instruction Format

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>cc</th>
<th>Contents</th>
<th>conditions (FCC field)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBGE:D</td>
<td>1010</td>
<td>Branch Greater or Equal</td>
<td>G or E</td>
</tr>
<tr>
<td>FBL:D</td>
<td>0100</td>
<td>Branch Less</td>
<td>L</td>
</tr>
<tr>
<td>FBUGE:D</td>
<td>1011</td>
<td>Branch Unordered or Greater or Equal</td>
<td>U or G or E</td>
</tr>
<tr>
<td>FBUG:D</td>
<td>0011</td>
<td>Branch Unordered or Greater</td>
<td>G or U</td>
</tr>
<tr>
<td>FBLE:D</td>
<td>1100</td>
<td>Branch Less or Equal</td>
<td>L or E</td>
</tr>
<tr>
<td>FBG:D</td>
<td>0010</td>
<td>Branch Greater</td>
<td>G</td>
</tr>
<tr>
<td>FBULE:D</td>
<td>1101</td>
<td>Branch Unordered or Less or Equal</td>
<td>E or L or U</td>
</tr>
<tr>
<td>FBU:D</td>
<td>0001</td>
<td>Branch Unordered</td>
<td>U</td>
</tr>
<tr>
<td>FBO:D</td>
<td>1110</td>
<td>Branch Ordered</td>
<td>E or L or G</td>
</tr>
</tbody>
</table>

EIT Occurrence and Detection
No interrupt is detected.
FR81 Family

CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

7.67    FCMPs (Single Precision Floating Point Compare)

FRk and FRj are compared, and the result is reflected on the floating point condition code (FCC) of floating point control register (FCR).

- **Assembler Format**
  
  FCMPs   FRk, FRj

- **Operation**
  
  FRk - FRj
  
  The FCC is set as follows according to the comparison result.

<table>
<thead>
<tr>
<th>FCC</th>
<th>Comparison result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000H</td>
<td>FRk = FRj</td>
</tr>
<tr>
<td>0100H</td>
<td>FRk &lt; FRj</td>
</tr>
<tr>
<td>0010H</td>
<td>FRk &gt; FRj</td>
</tr>
<tr>
<td>0001H</td>
<td>FRk ? FRj (not possible to compare)</td>
</tr>
</tbody>
</table>

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0 0 0 0 0 1 1 1 1 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td>(n+2)</td>
<td>-</td>
</tr>
</tbody>
</table>
### EIT Occurrence and Detection

An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.

### Calculation result and exception flag

<table>
<thead>
<tr>
<th>FRk</th>
<th>FRj</th>
<th>+0</th>
<th>-0</th>
<th>+Norm</th>
<th>-Norm</th>
<th>+INF</th>
<th>-INF</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+0</td>
<td>EQ/(-)</td>
<td>L/(-)</td>
<td>G/(-)</td>
<td>L(-)</td>
<td>G(-)</td>
<td>UO/(-)</td>
<td>UO/V</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L(-)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Norm</td>
<td>G/(-)</td>
<td></td>
<td>*1</td>
<td></td>
<td></td>
<td>G/(-)</td>
<td>E/(-)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Norm</td>
<td>L/(-)</td>
<td></td>
<td>*1</td>
<td></td>
<td></td>
<td>L/(-)</td>
<td>E/(-)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+INF</td>
<td>G/(-)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E/(-)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-INF</td>
<td>L/(-)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*1: G/(-) or L/(-) or E/(-)
FRk is divided by FRj, and its result is stored in FRi.

- **Assembler Format**
  
  FDIVs   FRk, FRj, FRi

- **Operation**
  
  FRk / FRj → FRi

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  9 cycles

- **Instruction Format**

  MSB  |  LSB  
  ---  |  ---  
  (n+0) | 0 0 0 0 0 1 1 1 1 0 1 0 1 0 | 1 0 |
  (n+2) | FRk  | FRj  | FRi  |

- **EIT Occurrence and Detection**
  
  An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.
### Calculation result and exception flag

<table>
<thead>
<tr>
<th>FRk</th>
<th>+0</th>
<th>-0</th>
<th>+Norm</th>
<th>-Norm</th>
<th>+INF</th>
<th>-INF</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>QNaN/V</td>
<td>+0(-)</td>
<td>-0/(-)</td>
<td>+0/(-)</td>
<td>-0/(-)</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0/(-)</td>
<td>+0/(-)</td>
<td>-0/(-)</td>
<td>+0/(-)</td>
<td>-0/(-)</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
<td></td>
</tr>
<tr>
<td>+Norm</td>
<td>INF/Z</td>
<td>-INF/Z</td>
<td>*1</td>
<td>*2</td>
<td>+0/(-)</td>
<td>-0/(-)</td>
<td>+0/(-)</td>
<td></td>
</tr>
<tr>
<td>-Norm</td>
<td>-INF/Z</td>
<td>INF/Z</td>
<td>*2</td>
<td>*1</td>
<td>-0/(-)</td>
<td>+0/(-)</td>
<td>+0/(-)</td>
<td></td>
</tr>
<tr>
<td>+INF</td>
<td>INF/(-&gt;)</td>
<td>-INF/(-&gt;)</td>
<td>+INF/(-&gt;)</td>
<td>-INF/(-&gt;)</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-INF</td>
<td>-INF/(-&gt;)</td>
<td>INF/(-&gt;)</td>
<td>-INF/(-&gt;)</td>
<td>+INF/(-&gt;)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN/V</td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*1: +INF/0, X or +0/U, X or +Norm/(X)

*2: -INF/0, X or -0/U, X or -Norm/(X)
FiTOs (Convert from Integer to Single Precision Floating Point)

A 32-bit signed integer in FRj is converted into a single-precision floating point value, and it is stored in FRi.

- **Assembler Format**
  
  FiTOs   FRj, FRi

- **Operation**
  
  (float) FRj → FRi

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

- **EIT Occurrence and Detection**
  
  An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.
## Calculation result and exception flag

<table>
<thead>
<tr>
<th>FRj</th>
<th>Output result</th>
</tr>
</thead>
<tbody>
<tr>
<td>±0</td>
<td>0/(-)</td>
</tr>
<tr>
<td>+Norm</td>
<td>+Norm/(X)</td>
</tr>
<tr>
<td>-Norm</td>
<td>-Norm/(X)</td>
</tr>
<tr>
<td>+MAX</td>
<td>+Norm/X</td>
</tr>
</tbody>
</table>
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

7.70  FLD (Single Precision Floating Point Data Load)

Loads the value at memory address Rj to FRi.

- Assembler Format
  
  FLD  @Rj, FRi

- Operation
  
  (Rj) → FRi

- Classification
  
  Single-precision floating point instruction, FR81 family

- Execution Cycles
  
  a cycle

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0) 0 0 0 0 1 1 1 1 1 1 0 0 Rj</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+2) - - - FRi</td>
</tr>
</tbody>
</table>

- EIT Occurrence and Detection
  
  A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.
7.71 FLD (Single Precision Floating Point Data Load)

Loads the value at memory address R13+Rj to FRi.

- **Assembler Format**
  
  FLD @(R13,Rj), FRi

- **Operation**
  
  (R13 + Rj) → FRi

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  a cycle

- **Instruction Format**

  MSB | LSB
  ---|---
  (n+0) 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | Rj
  (n+2) - | - | - | - | - | FRi

- **EIT Occurrence and Detection**

  A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.
7.72 FLD (Single Precision Floating Point Data Load)

Loads the value at memory address R14+o14 × 4 to FRi. Signed o14 value is calculated. The value in o14 × 4 is specified as disp16.

- **Assembler Format**
  
  FLD @(R14,disp16), FRi

- **Operation**
  
  \((R14 + o14 \times 4) \rightarrow FRi\)

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  a cycle

- **Instruction Format**

  MSB                                      LSB

  \[
  \begin{array}{cccccccccccc}
    \text{(n+0)} & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & \text{o14} \\
    \text{(n+2)} & & & & & & & & & & & & \text{FRi}
  \end{array}
  \]

- **EIT Occurrence and Detection**

  A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.
### 7.73 FLD (Single Precision Floating Point Data Load)

Loads the value at memory address R15+u14x4 to FRi. Unsigned u14 value is calculated. The value in u14x4 is specified as udisp16.

- **Assembler Format**
  
  \[
  \text{FLD @(R15,udisp16), FRi}
  \]

- **Operation**
  
  \[(R15 + u14 \times 4) \rightarrow \text{FRi}\]

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  a cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0 0 0 0 0 1 1 1 1 1 0 0 1 0 1 u14</td>
</tr>
<tr>
<td>(n+2)</td>
<td>u14</td>
</tr>
</tbody>
</table>

- **EIT Occurrence and Detection**

  A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.
7.74 FLD (Single Precision Floating Point Data Load)

Loads the value at memory address R15 to FRi, and adds 4 to R15.

- **Assembler Format**
  
  \[
  \text{FLD } @\text{R15}+, \text{ FRi}
  \]

- **Operation**
  
  \[
  (\text{R15}) \rightarrow \text{FRi} \\
  \text{R15} + 4 \rightarrow \text{R15}
  \]

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  a cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0 0 0 0 0 1 1 1 1 1 0 1 1 0 -</td>
</tr>
<tr>
<td>(n+2)</td>
<td>-</td>
</tr>
</tbody>
</table>

- **EIT Occurrence and Detection**

  A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.
7.75  FLD (Load Word Data in Memory to Floating Register)

Loads the word data at memory address BP+u16 × 4 to FRi. Unsigned u16 value is calculated. The value in u16 × 4 is specified as udisp18.

- Assembler Format

  FLD @(BP, udisp18), FRi

- Operation

  (BP+u16 × 4) → FRi

- Flag Change

  N, Z, V, C: Unchanged.

- Classification

  Memory load instruction, FR81 family

- Execution Cycles

  a cycle

- Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0 0 0 0 1 1 1 0 1 1 1 FRi</td>
</tr>
<tr>
<td>(n+2)</td>
<td>u16</td>
</tr>
</tbody>
</table>

- EIT Occurrence and Detection

  A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.
The registers shown on the frlist are sequentially restored from the stack. Registers FR0 to FR15 can be set on the frlist. They are processed in ascending order of register numbers.

- **Assembler Format**
  
  FLDM (frlist)

- **Operation**
  
  The following operations are repeated according to the number of registers specified in the parameter frlist.
  
  
  (R15) → FRi
  
  R15 + 4 → R15

  The bit and register relation of frlist of FLDM instruction is shown in Table 7.76-1.

<table>
<thead>
<tr>
<th>bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

- **Classification**
  
  Floating point instruction, FR81 family

- **Execution Cycles**
  
  na cycle (n: Transfer register number)
FR81 Family

● Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 1 1 1 1 1 0 1 1 1 -</td>
<td></td>
</tr>
<tr>
<td>(n+2)</td>
<td>frlist</td>
</tr>
</tbody>
</table>

● EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

If an exception occurs during repetition, the access that generated the exception is interrupted. The remaining values of frlist are stored in the RL of exception status register (ESR), and the exception process is executed.
FRk is multiplied by FRj, and FRi is added to its result and then stored in FRi.

- **Assembler Format**
  
  FMADDs  FRk, FRj, FRi

- **Operation**
  
  FRk \times FRj + FRi \rightarrow FRi

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  4 cycles

- **Instruction Format**
  
  MSB | LSB
  
  \( (n+0) \)  | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1
  
  \( (n+2) \)  | - | FRk | FRj | FRi

- **EIT Occurrence and Detection**
  
  An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.
**FR81 Family**

- **Calculation result and exception flag**

### [Multiplication]

<table>
<thead>
<tr>
<th>FRj</th>
<th>+0</th>
<th>-0</th>
<th>+Norm</th>
<th>-Norm</th>
<th>+INF</th>
<th>-INF</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRk:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>+0/(−)</td>
<td>-0/(−)</td>
<td>+0/(−)</td>
<td>-0/(−)</td>
<td>QNaN/V</td>
<td>QNaN/(−)</td>
<td>QNaN/V</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0/(−)</td>
<td>+0/(−)</td>
<td>-0/(−)</td>
<td>+0/(−)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Norm</td>
<td>+0/(−)</td>
<td>-0/(−)</td>
<td>*1</td>
<td>*2</td>
<td>+INF/(−)</td>
<td>-INF/(−)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Norm</td>
<td>-0/(−)</td>
<td>+0/(−)</td>
<td>*2</td>
<td>*1</td>
<td>-INF/(−)</td>
<td>+INF/(−)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+INF</td>
<td>QNaN/V</td>
<td>+INF/(−)</td>
<td>-INF/(−)</td>
<td>+INF/(−)</td>
<td>-INF/(−)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-INF</td>
<td>-INF/(−)</td>
<td>+INF/(−)</td>
<td>-INF/(−)</td>
<td>+INF/(−)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*1: +INF/0, X or +0/U, X or +Norm/(X)  
*2: -INF/0, X or -0/U, X or -Norm/(X)

### [Addition]

<table>
<thead>
<tr>
<th>FRj</th>
<th>+0</th>
<th>-0</th>
<th>+Norm</th>
<th>-Norm</th>
<th>+INF</th>
<th>-INF</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRk:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>+0/(−)</td>
<td>+Norm/(X)</td>
<td>-Norm/(X)</td>
<td>+INF/(−)</td>
<td>-INF/(−)</td>
<td>QNaN/(−)</td>
<td>QNaN/V</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0/(−)</td>
<td>-0/(−)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Norm</td>
<td>+Norm/(X)</td>
<td>*1</td>
<td>*2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Norm</td>
<td>-Norm/(X)</td>
<td>*2</td>
<td>*1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+INF</td>
<td>QNaN/V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-INF</td>
<td>-INF/(−)</td>
<td>QNaN/V</td>
<td>+INF/(−)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*1: +INF/0, X or -INF/P, X or ±Norm/(X)  
*2: ± 0/(−) or ± 0/U or ± Norm/(X)
7.78 FMOVs (Single Precision Floating Point Move)

Loads the value in FRj to FRi.

- Assembler Format
  
  FMOVs FRj, FRi

- Operation
  
  FRj → FRi

- Classification
  
  Single-precision floating point instruction, FR81 family

- Execution Cycles
  
  1 cycle

- Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0 0 0 0 1 1 1 1 0 1 0 1 1 1 0</td>
</tr>
<tr>
<td>(n+2)</td>
<td>-     -   FRj    FRi</td>
</tr>
</tbody>
</table>

- EIT Occurrence and Detection

  An invalid instruction exception (FPU absence error), an interrupt is detected.
FR81 Family

7.79 FMSUBs (Single Precision Floating Point Multiply and Subtract)

FRk is multiplied by FRj, and its result is subtracted by FRi and then stored in FRi.

- Assembler Format
  FMSUBs FRk, FRj, FRi

- Operation
  FRk \times FRj - FRi \rightarrow FRi

- Classification
  Single-precision floating point instruction, FR81 family

- Execution Cycles
  4 cycles

- Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (n+0) )</td>
<td>0 0 0 0 0 1 1 1 1 0 0 1 0 1 1 0</td>
</tr>
<tr>
<td>( (n+2) )</td>
<td>-</td>
</tr>
</tbody>
</table>

- EIT Occurrence and Detection
  An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.
### Calculation result and exception flag

#### [Multiplication]

<table>
<thead>
<tr>
<th>FRj</th>
<th>FRk</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0/(-)</td>
</tr>
<tr>
<td>-0</td>
<td>-0/(-)</td>
</tr>
<tr>
<td>+Norm</td>
<td>+0/(-)</td>
</tr>
<tr>
<td>-Norm</td>
<td>-0/(-)</td>
</tr>
<tr>
<td>+INF</td>
<td>QNaN/V</td>
</tr>
<tr>
<td>-INF</td>
<td>QNaN/V</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN/V</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN/V</td>
</tr>
</tbody>
</table>

*1: $+\text{INF}/0$, $X$ or $+\text{Norm}/(X)$

*2: $-\text{INF}/0$, $X$ or $-\text{Norm}/(X)$

#### [Addition]

<table>
<thead>
<tr>
<th>FRj</th>
<th>FRk</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0/(-)</td>
</tr>
<tr>
<td>-0</td>
<td>-0/(-)</td>
</tr>
<tr>
<td>+Norm</td>
<td>+Norm/(X)</td>
</tr>
<tr>
<td>-Norm</td>
<td>-Norm/(X)</td>
</tr>
<tr>
<td>+INF</td>
<td>QNaN/V</td>
</tr>
<tr>
<td>-INF</td>
<td>QNaN/V</td>
</tr>
<tr>
<td>QNaN</td>
<td>QNaN/V</td>
</tr>
<tr>
<td>SNaN</td>
<td>QNaN/V</td>
</tr>
</tbody>
</table>

*1: $+\text{INF}/0$ or $-\text{INF}/0$ or $\pm\text{Norm}/(X)$

*2: $\pm 0/(-)$ or $\pm 0/U$ or $\pm\text{Norm}/(X)$
7.80 FMULs (Single Precision Floating Point Multiply)

FRk is multiplied by FRj, and its result is stored in FRi.

- **Assembler Format**
  
  FMULs  FRk, FRj, FRi

- **Operation**
  
  FRk × FRj → FRi

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0 0 0 0 0 1 1 1 1 1 0 1 0 0 1 1 1 1</td>
</tr>
<tr>
<td>(n+2)</td>
<td></td>
</tr>
</tbody>
</table>

- **EIT Occurrence and Detection**

  An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.
### Calculation result and exception flag

<table>
<thead>
<tr>
<th>FRk</th>
<th>+0</th>
<th>-0</th>
<th>+Norm</th>
<th>-Norm</th>
<th>+INF</th>
<th>-INF</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0/(-)</td>
<td>-0/(-)</td>
<td>+0/(-)</td>
<td>-0/(-)</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
</tr>
<tr>
<td>-0</td>
<td>-0/(-)</td>
<td>+0/(-)</td>
<td>-0/(-)</td>
<td>+0/(-)</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
</tr>
<tr>
<td>+Norm</td>
<td>+0/(-)</td>
<td>-0/(-)</td>
<td>*1</td>
<td>*2</td>
<td>+INF/(-)</td>
<td>-INF/(-)</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
</tr>
<tr>
<td>-Norm</td>
<td>-0/(-)</td>
<td>+0/(-)</td>
<td>*2</td>
<td>*1</td>
<td>-INF/(-)</td>
<td>+INF/(-)</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
</tr>
<tr>
<td>+INF</td>
<td>QNaN/V</td>
<td>+INF/(-)</td>
<td>-INF/(-)</td>
<td>+INF/(-)</td>
<td>-INF/(-)</td>
<td>+INF/(-)</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
</tr>
<tr>
<td>-INF</td>
<td>-INF/(-)</td>
<td>+INF/(-)</td>
<td>-INF/(-)</td>
<td>+INF/(-)</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
<td>QNaN/V</td>
</tr>
</tbody>
</table>

*1: +INF/0, X or +0/U, X or +Norm/(X)

*2: -INF/0, X or 0/U, X or -Norm/(X)
7.81 FNEGs (Single Precision Floating Point sign reverse)

A sign of FRj value is inverted, and the result is stored in FRi.

- **Assembler Format**
  
  \[
  \text{FNEGs} \quad \text{FRj, FRi}
  \]

- **Operation**
  
  \[
  \text{FRj} \times -1 \rightarrow \text{FRi}
  \]

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
  \[
  \begin{array}{cccccccccccc}
  \text{MSB} & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
  \text{LSB} & \text{(n+0)} & \text{(n+2)} & - & - & \text{FRj} & \text{FRi}
  \end{array}
  \]

- **EIT Occurrence and Detection**
  
  An invalid instruction exception (FPU absence error), an interrupt is detected.
A square root of FRj is calculated, and its result is stored in FRI.

- **Assembler Format**
  
  FSQRTs FRj, FRI

- **Operation**
  
  \( \sqrt{FRj} \rightarrow FRI \)

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  14 cycles

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **EIT Occurrence and Detection**
  
  An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.

- **Calculation result and exception flag**

<table>
<thead>
<tr>
<th>FRj</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
</tr>
<tr>
<td>-0</td>
</tr>
<tr>
<td>+Norm</td>
</tr>
<tr>
<td>-Norm</td>
</tr>
<tr>
<td>+INF</td>
</tr>
<tr>
<td>-INF</td>
</tr>
<tr>
<td>QNaN</td>
</tr>
<tr>
<td>SNaN</td>
</tr>
<tr>
<td>+0/(-)</td>
</tr>
<tr>
<td>-0/(-)</td>
</tr>
<tr>
<td>+Norm/(X)</td>
</tr>
<tr>
<td>QNaN/V</td>
</tr>
<tr>
<td>+INF/(-)</td>
</tr>
<tr>
<td>QNaN/V</td>
</tr>
<tr>
<td>QNaN/(-)</td>
</tr>
<tr>
<td>QNaN/V</td>
</tr>
</tbody>
</table>
7.83 FST (Single Precision Floating Point Data Store)

Loads the value in FRi to memory address Rj.

- **Assembler Format**
  
  FST   FRi, @Rj

- **Operation**
  
  FRi → (Rj)

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  a cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0</td>
</tr>
<tr>
<td>(n+2)</td>
<td>-</td>
</tr>
</tbody>
</table>

- **EIT Occurrence and Detection**

  A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.
7.84 FST (Single Precision Floating Point Data Store)

Loads the value in FRi to memory address R13+Rj.

- **Assembler Format**
  
  FST FRi, @(R13,Rj)

- **Operation**
  
  FRi → (R13 + Rj)

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  a cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0</td>
</tr>
<tr>
<td>(n+2)</td>
<td>-</td>
</tr>
</tbody>
</table>

- **EIT Occurrence and Detection**

  A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.
7.85 FST (Single Precision Floating Point Data Store)

Loads the value in FRi to memory address R14+o14 × 4. Signed o14 value is calculated. The value in o14 × 4 is specified as disp16.

- **Assembler Format**
  
  FST FRi, @(R14,disp16)

- **Operation**
  
  FRi → (R14 + o14 × 4)

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  a cycle

- **Instruction Format**

  ![Instruction Format Diagram]

  **EIT Occurrence and Detection**

  A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

7.86  FST (Single Precision Floating Point Data Store)

FST (Single Precision Floating Point Data Store)

Loads the value in FRi to memory address R15+u14 × 4. Unsigned u14 value is calculated. The value in u14 × 4 is specified as udisp16.

- Assembler Format
  
  FST   FRi, @(R15,udisp16)

- Operation
  
  FRi → (R15 + u14 × 4)

- Classification
  
  Single-precision floating point instruction, FR81 family

- Execution Cycles
  
  a cycle

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>u14</td>
<td></td>
</tr>
<tr>
<td>(n+0)</td>
<td></td>
</tr>
<tr>
<td>(n+2)</td>
<td></td>
</tr>
</tbody>
</table>

- EIT Occurrence and Detection
  
  A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.
7.87 FST (Single Precision Floating Point Data Store)

R15 is subtracted by 4, and the value in FRi is loaded to the memory address identified by new R15.

- **Assembler Format**
  
  `FST FRi, @-R15`

- **Operation**
  
  `R15 - 4 \rightarrow R15`
  
  `FRi \rightarrow (R15)`

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  a cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0</td>
</tr>
<tr>
<td>(n+2)</td>
<td>-</td>
</tr>
</tbody>
</table>

- **EIT Occurrence and Detection**
  
  A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.
7.88 FST (Store Word Data in Floating Point Register to Memory)

Loads the word data in FRi to memory address BP+u16 × 4. Unsigned u16 value is calculated. The value in u16 × 4 is specified as udisp18.

- Assembler Format
  
  FST   FRi, @(BP, udisp18)

- Operation
  
  FRi → (BP+u16 × 4)

- Flag Change
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification
  
  Memory store instruction, FR81 family

- Execution Cycles
  
  a cycle

- Instruction Format
  
  \[
  \begin{array}{cccccccccc}
  \text{MSB} & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & F Ri \\
  (n+0) &  &  &  &  &  &  &  &  &  &  &  \\
  \hline
  (n+2) &  &  &  &  &  &  &  &  &  &  &  \\
  \text{LSB} &  &  &  &  &  &  &  &  &  &  &  \\
  \text{u16} &  &  &  &  &  &  &  &  &  &  \\
  \end{array}
  \]

- EIT Occurrence and Detection
  
  A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.
7.89 FSTM (Single Precision Floating Point Data Store from Multiple Register)

The registers shown on frlist are sequentially saved in the stack. Registers FR0 to FR15 can be set on the frlist. They are processed in descending order of register numbers.

- **Assembler Format**
  
  FSTM (frlist)

- **Operation**
  
  The following operations are repeated according to the number of registers specified in the parameter frlist.
  
  R15 - 4 \(\rightarrow\) R15

  (R15) \(\rightarrow\) FRi

  The bit and register relation of frlist of FSTM instruction is shown in Table 7.89-1.

<table>
<thead>
<tr>
<th>bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  na cycle (n: Transfer register number)

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(n+0)</td>
<td></td>
</tr>
<tr>
<td>(n+2)</td>
<td></td>
</tr>
</tbody>
</table>
EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

If an exception occurs during repetition, the access that generated the exception is interrupted. The remaining values of frlist are stored in the register list (RL) of exception status register (ESR), and the exception process is executed.
A single-precision floating point value of FRj is converted into a 32-bit signed integer, and it is stored in FRi.

- **Assembler Format**
  
  FsTOi   FRj, FRi

- **Operation**
  
  (int) FRj → FRi

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0 0 0 0 0 1 1 1 1 1 0 1 0 1 0 0 1</td>
</tr>
<tr>
<td>(n+2)</td>
<td>FRj</td>
</tr>
<tr>
<td></td>
<td>FRi</td>
</tr>
</tbody>
</table>

- **EIT Occurrence and Detection**

  An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.
### Calculation result and exception flag

<table>
<thead>
<tr>
<th>FRj</th>
<th>Output result</th>
</tr>
</thead>
<tbody>
<tr>
<td>± 0</td>
<td>0/(-)</td>
</tr>
<tr>
<td>± Den</td>
<td>0/D</td>
</tr>
<tr>
<td>+Norm</td>
<td>+0/(X), +Norm/(X), +MAX/V</td>
</tr>
<tr>
<td>-Norm</td>
<td>+0/(X), -Norm/(X), -MAX/V</td>
</tr>
<tr>
<td>+INF</td>
<td>+MAX/V</td>
</tr>
<tr>
<td>-INF</td>
<td>-MAX/V</td>
</tr>
<tr>
<td>QNaN, SNaN</td>
<td>± MAX/V</td>
</tr>
</tbody>
</table>
7.91 FSUBs (Single Precision Floating Point Subtract)

FRk is subtracted by FRj, and its result is stored in FRi.

- **Assembler Format**
  
  
  FSUBs  FRk, FRj, FRi

- **Operation**
  
  FRk - FRj → FRi

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0</td>
</tr>
<tr>
<td>(n+2)</td>
<td></td>
</tr>
</tbody>
</table>

- **EIT Occurrence and Detection**

  An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.
## 7.91 Calculation result and exception flag

<table>
<thead>
<tr>
<th>FRk</th>
<th>+0</th>
<th>-0</th>
<th>+Norm/X</th>
<th>-Norm/X</th>
<th>+INF/X</th>
<th>-INF/X</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0/(-)</td>
<td>+Norm/X</td>
<td>-Norm/X</td>
<td>+INF/X</td>
<td>-INF/X</td>
<td>QNaN/(-)</td>
<td>QNaN/V</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0/(-)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN/V</td>
<td></td>
</tr>
<tr>
<td>+Norm</td>
<td>+Norm/X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN/V</td>
<td></td>
</tr>
<tr>
<td>-Norm</td>
<td>-Norm/X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN/V</td>
<td></td>
</tr>
<tr>
<td>+INF</td>
<td>+INF/(-)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN/V</td>
<td></td>
</tr>
<tr>
<td>-INF</td>
<td>-INF/(-)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN/V</td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN/V</td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNaN/V</td>
<td></td>
</tr>
</tbody>
</table>

*1: +INF/0 or -INF/0 or ± Norm/(X)
*2: ± 0/(-) or ± 0/U or ± Norm/(X)
This is a software interrupt instruction. Reads the vector table for the interrupt vector number u8 to determine the branch destination address, and branches.

- **Assembler Format**
  
  ```
  INT #u8
  ```

  Vector numbers 9 to 13, 64 and 65 are used by emulators for debugging interrupts and therefore the corresponding numbers "INT #9" to "INT #13", "INT #64", "INT #65" should not be used in user programs.

- **Operation**
  
  ```
  SSP-4 → SSP
  PS → (SSP)
  SSP-4 → SSP
  PC+2 → (SSP)
  "0" → CCR:I
  "0" → CCR:S
  (TBR+3FCH-u8 × 4) → PC
  ```

  Stores the values of the program counter (PC) and program status (PS) to the stack indicated by the system stack pointer (SSP) for interrupt processing. Writes "0" to the S flag in the condition code register (CCR), and uses the SSP as the stack pointer for following steps. Writes "0" to the I flag (interrupt enable flag) in the CCR to disable external interrupts. Reads the vector table for the interrupt vector number u8 to determine the branch destination address, and branches.

- **Flag Change**

<table>
<thead>
<tr>
<th>S</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.
  S, I: Cleared.

- **Classification**

  Non-Delayed Branching instruction
7.92

- **Execution Cycles**
  
  1+3a cycles

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>u8</td>
</tr>
</tbody>
</table>

- **Execution Example**

  ```
  INT #20H ; Bit pattern of the instruction: 0001 1111 0010 0000
  ```

  Before execution

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>000FFF7C 68096800</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>7FFFFFF8 xxxxxxxxxx</td>
</tr>
<tr>
<td>7FFFFFFFC xxxxxxxxxx</td>
</tr>
<tr>
<td>80000000 xxxxxxxxxx</td>
</tr>
</tbody>
</table>

  After execution

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>000FFF7C 68096800</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>7FFFFFF8 80888088</td>
</tr>
<tr>
<td>7FFFFFFFC FFFFF8F0</td>
</tr>
<tr>
<td>80000000 xxxxxxxxxx</td>
</tr>
</tbody>
</table>
This software interrupt instruction is used for debugging. It determines the branch destination address by reading interrupt vector number "#9" from the vector table, then branches.

- **Assembler Format**
  
  INTE

- **Operation**
  
  SSP-4 → SSP
  PS → (SSP)
  SSP-4 → SSP
  PC+2 → (SSP)
  4 → ILM
  "0" → CCR:S
  (TBR+3D8H) → PC

  It stores the values of the program counter (PC) and program status (PS) to the stack indicated by the system stack pointer (SSP) for interrupt processing. It writes "0" to the S flag in the condition code register (CCR), and uses the SSP as the stack pointer for the following steps. It determines the branch destination address by reading interrupt vector number #9 from the vector table, then branches.

  There is not change to the I flag in the condition code register (CCR). The interrupt level mask register (ILM) in the program status (PS) is set to level 4.

- **Flag Change**
  
<table>
<thead>
<tr>
<th>S</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  S: Cleared.

- **Classification**
  
  Non-Delayed Branching instruction
7.93 Execution Cycles

1+3a cycles

Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Execution Example

INTE ; Bit pattern of the instruction: 1001 1111 0011 0000

Before execution

<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>000FFD8</td>
<td>000FFD8</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>7FFFFFFF8</td>
<td>7FFFFFFF8</td>
</tr>
<tr>
<td>7FFFFFC</td>
<td>80000000</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>000FFD8</td>
<td>000FFD8</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>7FFFFFFF8</td>
<td>7FFFFFFF8</td>
</tr>
<tr>
<td>7FFFFFC</td>
<td>80000000</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FR81 Family

7.94 JMP (Jump)

This is a branching instruction without a delay slot. Branches to the address indicated in Ri.

- Assembler Format
  
  JMP @Ri

- Operation
  
  Ri → PC

- Flag Change
  
  N, Z, V, C: Unchanged.

- Classification
  
  Non-Delayed Branching instruction

- Execution Cycles
  
  2 cycles

- Instruction Format
  
  \[
  \begin{array}{cccccccccccc}
  \text{MSB} & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & \text{Ri} \\
  \text{LSB} & & & & & & & & & & & &
  \end{array}
  \]
● Execution Example

JMP @R1 ; Bit pattern of the instruction: 1001 0111 0000 0001

Before execution

<table>
<thead>
<tr>
<th>R1</th>
<th>0000 8000</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>FF 80 0000</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R1</th>
<th>0000 00FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>C000 8000</td>
</tr>
</tbody>
</table>

(PC)
This is a branching instruction with delay slot. Branches to the address indicated by Ri.

- **Assembler Format**
  
  ```assembler
  JMP:D   @Ri
  ```

- **Operation**
  
  ```text
  Ri → PC
  ```

- **Flag Change**
  
  ```table
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
  ```

  N, Z, V, C: Unchanged.

- **Classification**
  
  Delayed Branching instruction

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
  ```plaintext
  MSB  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
  1    | 0   | 0   | 1   | 1   | 1   | 1   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | Ri
  ```
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

7.95

FR81 Family

Execution Example

JMP:D @R1 ; Bit pattern of the instruction: 1001 1111 0000 0001

LDI:8  #0FFH, R1 ; Instruction placed in delay slot

The instruction placed in the delay slot will be executed before execution of the branch destination instruction. The value R1 above will vary according to the specifications of the LDI:8 instruction placed in the delay slot.

Before execution of JMP instruction

R1 | C 0 0 0 8 0 0 0
PC | F F 8 0 0 0 0 0

After branching

R1 | 0 0 0 0 0 0 F F
PC | C 0 0 0 8 0 0 0
FR81 Family

7.96 LCALL (Long Call Subroutine)

This is a branching instruction without a delay slot. The next instruction address is stored in the return pointer (RP), and then control branches to the address identified by label21 relative to the program counter (PC). The value in rel20 is doubled during address calculation, and its sign is extended.

- Assembler Format
  LCALL label21

- Operation
  PC + 4 \rightarrow RP
  PC + 4 + exts(rel20 \times 2) \rightarrow PC

- Flag Change
  N, Z, V, C: Unchanged.

- Classification
  Branching instruction without delay, FR81 family

- Execution Cycles
  2 cycles

- Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0 0 0 0 0 1 1 1 0 0 1 0</td>
</tr>
<tr>
<td>rel20 (Upper)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(n+2)</th>
<th>rel20 (Lower)</th>
</tr>
</thead>
</table>

- EIT Occurrence and Detection
  An interrupt is detected.
7.97  LCALL:D (Long Call Subroutine)

This is a branching instruction with a delay slot. The next instruction address is stored in the return pointer (RP), and then control branches to the address identified by label21 relative to the program counter (PC). The value in rel20 is doubled during address calculation, and its sign is extended.

- **Assembler Format**
  
  LCALL:D label21

- **Operation**
  
  \[ \text{PC} + 6 \rightarrow \text{RP} \]
  
  \[ \text{PC} + 4 + \text{exts}(\text{rel20} \times 2) \rightarrow \text{PC} \]

- **Flag Change**
  
  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  - & - & - & - \\
  \end{array}
  \]

  N, Z, V, C: Unchanged.

- **Classification**
  
  Branching instruction with delay, FR81 family

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>rel20 (Upper)</td>
</tr>
<tr>
<td>0 0 0 1 0 1 1 1 0 0 1 0</td>
<td></td>
</tr>
</tbody>
</table>

  | (n+2) | rel20 (Lower) |
  |       |               |

- **EIT Occurrence and Detection**

  No interrupt is detected.
7.98 LD (Load Word Data in Memory to Register)

Loads the word data at memory address Rj to Ri.

- **Assembler Format**
  
  \[
  \text{LD } \mathbf{\text{@Rj}}, \text{Ri}
  \]

- **Operation**
  
  \[(\mathbf{\text{Rj}}) \rightarrow \text{Ri}\]

- **Flag Change**
  
  \[
  \begin{array}{c|c|c|c|c}
  \hline
  \text{N} & \text{Z} & \text{V} & \text{C} \\
  \hline
  - & - & - & - \\
  \end{array}
  \]

  N, Z, V, C: Unchanged.

- **Classification**
  
  Memory Load instruction, Instruction with delay slot

- **Execution Cycles**
  
  b cycle

- **Instruction Format**
  
  \[
  \begin{array}{cccccccc}
  \text{MSB} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{Rj} & \text{Ri} & \text{LSB}
  \end{array}
  \]
Execution Example

LD @R2, R3 ; Bit pattern of the instruction: 0000 0100 0010 0011

Before execution

| R2 | 1 2 3 4 5 6 7 8 |
| R3 | 0 0 0 0 0 0 0 0 |

| Memory |
| 12345678 |
| 8 7 6 5 4 3 2 1 |

After execution

| R2 | 1 2 3 4 5 6 7 8 |
| R3 | 8 7 6 5 4 3 2 1 |

| Memory |
| 12345678 |
| 8 7 6 5 4 3 2 1 |
7.99  LD (Load Word Data in Memory to Register)

Loads the word data at memory address R13 + Rj to Ri.

- Assembler Format
  \[
  \text{LD} \quad @(R13, Rj), Ri
  \]

- Operation
  \[
  (R13+Rj) \rightarrow Ri
  \]

- Flag Change
  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  - & - & - & - \\
  \end{array}
  \]
  N, Z, V, C: Unchanged.

- Classification
  Memory Load instruction, Instruction with delay slot

- Execution Cycles
  b cycle

- Instruction Format
  \[
  \begin{array}{ccccccc}
  \text{MSB} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{Rj} & \text{Ri} \\
  \text{LSB} & & & & & & & & & & & &
  \end{array}
  \]
● Execution Example

LD @(R13, R2), R3 ; Bit pattern of the instruction: 0000 0000 0010 0011

Before execution | After execution
---|---
R2  | 0 0 0 0 0 0 0 4
R3  | x x x x x x x x
R13 | 1 2 3 4 5 6 7 8

Memory

Before execution

1234567B | Memory | 1234567B | Memory
---|---|---|---
1234567C | 8 7 6 5 4 3 2 1 | 1234567C | 8 7 6 5 4 3 2 1
7.100  LD (Load Word Data in Memory to Register)

Loads the word data at memory address R14 + 08 × 4 to Ri. The value of 08 × 4 is specified as disp10.

- Assembler Format
  
  \[
  \text{LD } @(R14, \text{disp10}), \text{Ri}
  \]

- Operation
  
  \[
  (R14+08 \times 4) \rightarrow \text{Ri}
  \]

- Flag Change
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification
  
  Memory Load instruction, Instruction with delay slot

- Execution Cycles
  
  b cycle

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>08</td>
<td>Ri</td>
</tr>
</tbody>
</table>
Execution Example

LD @(R14,4), R3 ; Bit pattern of the instruction: 0010 0000 0001 0011

Before execution

<table>
<thead>
<tr>
<th>R3</th>
<th>R14</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x x x</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>1234567C</td>
</tr>
<tr>
<td>8 7 6 5 4 3 2 1</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R3</th>
<th>R14</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 7 6 5 4 3 2 1</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>1234567C</td>
</tr>
<tr>
<td>8 7 6 5 4 3 2 1</td>
</tr>
</tbody>
</table>
FR81 Family

7.101  LD (Load Word Data in Memory to Register)

 Loads the word data at memory address R15 + u4 × 4 to Ri. The value u4 is an unsigned calculation. The value of u4 × 4 is specified as udisp6.

- Assembler Format
  
  `LD @(R15, udisp6), Ri`

- Operation
  
  `(R15+u4 ×4) → Ri`

- Flag Change
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification
  
  Memory Load instruction, Instruction with delay slot

- Execution Cycles
  
  b cycle

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>u4</th>
<th>Ri</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>u4</td>
<td>Ri</td>
</tr>
</tbody>
</table>
7.101

Execution Example

LD @(R15,4), R3 ; Bit pattern of the instruction: 0000 0011 0011 0011

Before execution

R3  x x x x  x x x x
R15 1 2 3 4 5 6 7 8
12345678  Memory  12345678  Memory
1234567C  8 7 6 5 4 3 2 1  1234567C  8 7 6 5 4 3 2 1

After execution

R3  8 7 6 5 4 3 2 1
R15 1 2 3 4 5 6 7 8
12345678  Memory  12345678  Memory
1234567C  8 7 6 5 4 3 2 1  1234567C  8 7 6 5 4 3 2 1
7.102 LD (Load Word Data in Memory to Register)

Loads the word data at memory address R15 to Ri, and adds 4 to the value of R15. If R15 is given as parameter Ri, the value read from the memory will be loaded into memory address R15.

- Assembler Format

  \[
  \text{LD} \  @R15+, \ Ri
  \]

- Operation

  \[
  (R15) \rightarrow Ri \\
  R15 + 4 \rightarrow R15
  \]

- Flag Change

  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  \hline
  - & - & - & -
  \end{array}
  \]

  N, Z, V, C: Unchanged.

- Classification

  Memory Load instruction, Instruction with delay slot

- Execution Cycles

  b cycle

- Instruction Format

  MSB
  
  | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | Ri |
  
  LSB
Execution Example

LD @R15+, R3 ; Bit pattern of the instruction: 0000 0111 0000 0011

Memory

Before execution

R3 | R15 | Memory
---|-----|---
xx | 12345678 | 12345678
xx | 1234567C | 1234567C

After execution

R3 | R15 | Memory
---|-----|---
87654321 | 12345678 | 12345678
87654321 | 1234567C | 1234567C
7.103   LD (Load Word Data in Memory to Register)

Loads the word data at memory address BP+u16 × 4 to Ri. Unsigned u16 value is calculated. The value in u16 × 4 is specified as udisp18.

- Assembler Format
  
  LD @(BP, udisp18), Ri

- Operation
  
  (BP+u16 × 4) → Ri

- Flag Change
  

- Classification
  
  Memory load instruction, FR81 family

- Execution Cycles
  
  a cycle

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0 0 0 0 1 1 1 0 1 0 0 Ri</td>
</tr>
<tr>
<td>(n+2)</td>
<td>u16</td>
</tr>
</tbody>
</table>

- EIT Occurrence and Detection
  
  A data access protection violation exception, an invalid instruction exception (a data access error), or an interrupt is detected.
7.104 LD (Load Word Data in Memory to Register)

Loads the word data at memory address R15 to dedicated register Rs, and adds 4 to the value of R15.

- Assembler Format
  
  \[ \text{LD} \ @R15+, \ Rs \]

- Operation
  
  \( (R15) \rightarrow Rs \)
  
  \( R15 + 4 \rightarrow R15 \)

  If TBR, SSP, or ESR is specified in user mode or if a non-existing register number is specified in user mode, an invalid instruction exception (system-only register access) is generated.

  There is no restriction in privilege mode. If a number without a dedicated register is specified for Rs in privilege mode, a value being read from memory is abandoned.

  If Rs is designated as the system stack pointer (SSP) or user stack pointer (USP), and that pointer is indicating R15 (the S flag in the condition code register (CCR) is set to "0" to indicate the SSP, and to "1" to indicate the USP), the last value remaining in R15 will be the value read from memory.

- Flag Change

  \[ \begin{array}{c|c|c|c|c} 
    N & Z & V & C \\
    \hline
    - & - & - & - 
  \end{array} \]

  N, Z, V, C: Unchanged.

- Classification

  Memory Load instruction, Instruction with delay slot, FR81 updating

- Execution Cycles

  b cycle
FR81 Family

● Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>Rs</td>
</tr>
</tbody>
</table>

● EIT Occurrence and Detection

User mode:
An invalid instruction exception (system-only register access) is generated. A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected.

Privilege mode:
A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected.

● Execution Example

LD @R15+, MDH ; Bit pattern of the instruction: 0000 0111 1000 0100

Before execution

<table>
<thead>
<tr>
<th>R15</th>
<th>MDH</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345670</td>
<td>x x x x</td>
<td>12345670</td>
</tr>
<tr>
<td>12345674</td>
<td>x x x x</td>
<td>12345674</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R15</th>
<th>MDH</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345670</td>
<td>8 7 6 5</td>
<td>12345670</td>
</tr>
<tr>
<td>12345674</td>
<td>8 7 6 5</td>
<td>12345674</td>
</tr>
</tbody>
</table>
7.105 LD (Load Word Data in Memory to Program Status Register)

Loads the word data at memory address R15 to the program status (PS), and adds 4 to the value of R15.

- **Assembler Format**
  ```markdown
  LD  @R15+, PS
  ```

- **Operation**
  
  \[
  (R15) \rightarrow PS \\
  R15 + 4 \rightarrow R15
  \]

  The contents of system status register (SSR) cannot be changed by this instruction regardless of the selected operation mode. If this instruction is executed in user mode, only the D1, D0, N, Z, V and C flags can be changed. The other flag values are not updated. Any bit other than SSR can be changed in privilege mode.

  At the time this instruction is executed, if the value of the interrupt level mask register (ILM) is in the range 16 to 31, only new ILM settings between 16 and 31 can be entered. If data in the range 0 to 15 is loaded from memory, the value 16 will be added to that data before being transferred to the ILM. IF the original ILM value is in the range 0 to 15, then any value from 0 to 31 can be transferred to the ILM.

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Data of (R15) is transferred.

- **Classification**
  
  Memory Load instruction, FR81 updating

- **Execution Cycles**
  
  1+a cycles
FR81 Family

- Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 1 1 1 0 0 1 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

- EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected. If the interrupt level mask register (ILM) or interrupt enable flag (I) is changed, an interrupt is detected using the changed values.

- Execution Example

LD @R15+, PS ; Bit pattern of the instruction: 0000 0111 1001 0000

![Before execution vs After execution diagram](image-url)
7.106 LDI:20 (Load Immediate 20bit Data to Destination Register)

Extends the 20-bit immediate data with 12 zeros in the higher bits, loads to Ri.

- **Assembler Format**
  
  LDI:20 #i20, Ri

- **Operation**
  
  \[ \text{extu(i20)} \rightarrow \text{Ri} \]

- **Flag Change**
  
  \[
  \begin{array}{cccc}
  \text{N} & \text{Z} & \text{V} & \text{C} \\
  - & - & - & - \\
  \end{array}
  \]
  
  N, Z, V, C: Unchanged.

- **Classification**
  
  Immediate Data Transfer instruction

- **Execution Cycles**
  
  2 cycles

- **Instruction Format**
  
  MSB
  \[
  \begin{array}{cccccccc}
  \text{(n+0)} & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & \text{i20 (higher)} & \text{Ri} \\
  \end{array}
  \]
  
  LSB
  \[
  \begin{array}{cccc}
  \text{(n+2)} & \text{i20 (lower)} \\
  \end{array}
  \]
FR81 Family

- Execution Example

LDI: 20 #54321H, R3
: Bit pattern of the instruction: 1001 1011 0101 0011
: 0100 0011 0010 0001

R3

0 0 0 0 0 0 0 0

Before execution

→

R3

0 0 0 5 4 3 2 1

After execution
7.107  LDI:32 (Load Immediate 32 bit Data to Destination Register)

Loads 1 word of immediate data to Ri.

- Assembler Format
  
  LDI:32  #i32, Ri

- Operation
  
  i32 → Ri

- Flag Change

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification

  Immediate Data Transfer instruction

- Execution Cycles

  d cycle

- Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>1 0 0 1 1 1 1 1 0 0 0 Ri</td>
</tr>
<tr>
<td>(n+2)</td>
<td>i32 (higher)</td>
</tr>
<tr>
<td>(n+4)</td>
<td>i32 (lower)</td>
</tr>
</tbody>
</table>
Execution Example

LDI:32 #87654321H, R3 ; Bit pattern of the instruction: 1001 1111 1000 0011
; 1000 0111 0110 0101
; 0100 0011 0010 0001

Before execution

R3  0 0 0 0 0 0 0 0

After execution

R3  8 7 6 5 4 3 2 1
7.108  **LDI:8 (Load Immediate 8bit Data to Destination Register)**

Extends the 8-bit immediate data with 24 zeros in the higher bits, loads to Ri.

- **Assembler Format**
  
  \[
  \text{LDI:8 } \#i8, \text{Ri}
  \]

- **Operation**
  
  \[
  \text{extu(i8)} \rightarrow \text{Ri}
  \]

- **Flag Change**
  
  \[
  \begin{array}{cccc}
  \text{N} & \text{Z} & \text{V} & \text{C} \\
  - & - & - & -
  \end{array}
  \]

  N, Z, V, C: Unchanged.

- **Classification**

  Immediate Data Transfer instruction, Instruction with delay slot

- **Execution Cycles**

  1 cycle

- **Instruction Format**

  \[
  \begin{array}{cccc|c|c}
  \text{MSB} & 1 & 1 & 0 & 0 & i8 & \text{Ri} \\
  \text{LSB}
  \end{array}
  \]
FR81 Family

Execution Example

LDI:8 #21H, R3 ; Bit pattern of the instruction: 1100 0010 0001 0011

R3 | 0 0 0 0 0 0 0 0 |
    | Before execution |

R3 | 0 0 0 0 0 0 0 2 1 |
    | After execution  |
7.109 LDM0 (Load Multiple Registers)

The LDM0 instruction stores the word data from the address R15 to the registers in the range R0 to R7 as members of the parameter reglist and repeats the operation of adding 4 to R15. Registers are processed in ascending numerical order.

- Assembler Format

```
LDM0  (reglist)
```

Registers from R0 to R7 are separated in reglist, multiple register are arranged and specified.

- Operation

The following operations are repeated according to the number of registers specified in the parameter reglist.

```
(R15) → Ri
R15+4 → R15
```

Bit values and register numbers for reglist (LDM0) are shown in Table 7.109-1.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>R7</td>
</tr>
<tr>
<td>6</td>
<td>R6</td>
</tr>
<tr>
<td>5</td>
<td>R5</td>
</tr>
<tr>
<td>4</td>
<td>R4</td>
</tr>
<tr>
<td>3</td>
<td>R3</td>
</tr>
<tr>
<td>2</td>
<td>R2</td>
</tr>
<tr>
<td>1</td>
<td>R1</td>
</tr>
<tr>
<td>0</td>
<td>R0</td>
</tr>
</tbody>
</table>

- Flag Change

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

N, Z, V, C: Unchanged.
FR81 Family

● Classification

Other instructions

● Execution Cycles

If "n" is the number of registers specified in the parameter reglist, the execution cycles required are as follows.

When n=0: 1 cycle
Otherwise: b × n cycles

● Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
<th>reglist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

● Execution Example

LDM0 (R3, R4)  ; Bit pattern of the instruction: 1000 1100 0001 1000

<table>
<thead>
<tr>
<th>R3</th>
<th>R4</th>
<th>R15</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x</td>
<td>x x x x</td>
<td>7 F F F F C 0</td>
</tr>
<tr>
<td>x x x x</td>
<td>x x x x</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>R4</td>
<td>R15</td>
</tr>
<tr>
<td>9 0 B C</td>
<td>8 3 4 3 8 3 4 A</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 F F F F C 8</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>7FFFFFC0</th>
<th>7FFFFFC0</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 0 B C 9 3 6 3</td>
<td>9 0 B C 9 3 6 3</td>
</tr>
<tr>
<td>7FFFFFC4</td>
<td>7FFFFFC4</td>
</tr>
<tr>
<td>8 3 4 3 8 3 4 A</td>
<td>8 3 4 3 8 3 4 A</td>
</tr>
<tr>
<td>7FFFFFC8</td>
<td>7FFFFFC8</td>
</tr>
<tr>
<td>x x x x x x x x</td>
<td>x x x x x x x</td>
</tr>
</tbody>
</table>

Before execution  After execution
7.110 LDM1 (Load Multiple Registers)

Loads the word data of address R15 to multiple registers R8 to R15 specified in reglist, repeats the operation of adding 4 to R15. Registers are processed in ascending numerical order. If R15 is specified in the parameter reglist, the final contents of R15 will be read from memory.

- Assembler Format

LDM1 (reglist)

Registers from R8 to R15 are separated in reglist, multiple register are arranged and specified.

- Operation

The following operations are repeated according to the number of registers specified in the parameter reglist.

(R15) → Ri
R15+4 → R15

Bit values and register numbers for reglist (LDM1) are shown in Table 7.110-1.

Table 7.110-1 Bit values and register numbers for reglist (LDM1)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>R15</td>
</tr>
<tr>
<td>6</td>
<td>R14</td>
</tr>
<tr>
<td>5</td>
<td>R13</td>
</tr>
<tr>
<td>4</td>
<td>R12</td>
</tr>
<tr>
<td>3</td>
<td>R11</td>
</tr>
<tr>
<td>2</td>
<td>R10</td>
</tr>
<tr>
<td>1</td>
<td>R9</td>
</tr>
<tr>
<td>0</td>
<td>R8</td>
</tr>
</tbody>
</table>

- Flag Change

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N, Z, V, C: Unchanged.
FR81 Family

● Classification

Other instructions

● Execution Cycles

If "n" is the number of registers specified in the parameter reglist the execution cycles required are as follows.
When n=0: 1 cycle
Otherwise: b × n cycles

● Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>reglist</th>
<th>LSB</th>
</tr>
</thead>
</table>

● Execution Example

LDM1 (R10, R11, R12) ; Bit pattern of the instruction: 1000 1101 0001 1100

<table>
<thead>
<tr>
<th>R10</th>
<th>x x x x x x x x</th>
<th>R10</th>
<th>8 F E 3 9 E 8 A</th>
</tr>
</thead>
<tbody>
<tr>
<td>R11</td>
<td>x x x x x x x x</td>
<td>R11</td>
<td>9 0 B C 9 3 6 3</td>
</tr>
<tr>
<td>R12</td>
<td>x x x x x x x x</td>
<td>R12</td>
<td>8 D F 7 8 8 E 4</td>
</tr>
<tr>
<td>R15</td>
<td>7 F F F F F C 0</td>
<td>R15</td>
<td>7 F F F F F C C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FFFFFC0</td>
</tr>
<tr>
<td>7FFFFFC4</td>
</tr>
<tr>
<td>7FFFFFC8</td>
</tr>
<tr>
<td>7FFFFFCC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FFFFFC0</td>
</tr>
<tr>
<td>7FFFFFC4</td>
</tr>
<tr>
<td>7FFFFFC8</td>
</tr>
<tr>
<td>7FFFFFCC</td>
</tr>
</tbody>
</table>

Before execution | After execution
7.111 LDUB (Load Byte Data in Memory to Register)

Extends with zeros the byte data at memory address Rj, loads to Ri.

- **Assembler Format**
  
  \[
  \text{LDUB} \ @ \text{Rj}, \text{Ri}
  \]

- **Operation**
  
  \[\text{extU}((\text{Rj})) \rightarrow \text{Ri}\]

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Memory Load instruction, Instruction with delay slot

- **Execution Cycles**
  
  \[b \text{ cycle}\]

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

  MSB: Most Significant Bit, LSB: Least Significant Bit.
**FR81 Family**

- **Execution Example**

LDUB @R2, R3 ; Bit pattern of the instruction: 0000 0110 0010 0011

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>x x x x x x x x</td>
</tr>
</tbody>
</table>

Memory

Before exec | After exec

12345678 | 2 1
12345678 | 2 1

Memory

Before execution | After execution
Extends with zeros the byte data at memory address R13 + Rj, loads to Ri.

- **Assembler Format**
  
  LDUB @(R13, Rj), Ri

- **Operation**
  
  extu((R13+Rj)) → Ri

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Memory Load instruction, Instruction with delay slot

- **Execution Cycles**
  
  b cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

FR81 Family

Execution Example

LDUB @(R13, R2), R3 ; Bit pattern of the instruction: 0000 0010 0010 0011

Before execution

R2
0 0 0 0 0 0 0 4
R3
x x x x x x x x
R13
1 2 3 4 5 6 7 8

12345678 12345678
Memory Memory
2 1 2 1

After execution

R2
0 0 0 0 0 0 0 4
R3
0 0 0 0 0 0 2 1
R13
1 2 3 4 5 6 7 8

12345678 12345678
Memory Memory
2 1 2 1
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

FR81 Family

7.113  LDUB (Load Byte Data in Memory to Register)

Extends with zeros the byte data at memory address 14 + o8, loads to Ri. The value o8 is a signed calculation. The value of o8 is specified in disp8.

- Assembler Format

  LDUB @(R14, disp8), Ri

- Operation

  extu((R14+o8)) → Ri

- Flag Change

  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification

  Memory Load instruction, Instruction with delay slot

- Execution Cycles

  b cycle

- Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>o8</td>
<td>Ri</td>
</tr>
</tbody>
</table>
FR81 Family

- Execution Example

LDUB @(R14,1), R3 ; Bit pattern of the instruction: 0110 0000 0001 0011

Before execution

R3: x x x x x x x x
R14: 1 2 3 4 5 6 7 8
Memory: 12345678

12345679: 2 1

After execution

R3: 0 0 0 0 0 0 2 1
R14: 1 2 3 4 5 6 7 8
Memory: 12345678

12345679: 2 1
7.114   LDUB (Load Byte Data in Memory to Register)

Loads the byte data at memory address BP+u16 to Ri. Unsigned u16 value is calculated. The value in u16 is specified as udisp16.

- Assembler Format
  
  ```
  LDUB @(BP, udisp16), Ri
  ```

- Operation
  
  ```
  (BP+u16) → Ri
  ```

- Flag Change
  
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification
  
  Memory Load instruction, FR81 family

- Execution Cycles
  
  a cycle

- Instruction Format
  
  ```
  MSB                                      LSB
  (n+0)  0   0   0   0   0   1   1   1   0   1   1   0   Ri
  (n+2)          u16
  ```

- EIT Occurrence and Detection
  
  A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected.
7.115 LDUH (Load Halfword Data in Memory to Register)

Extends with zeros the half-word data at memory address Rj, loads to Ri.

- **Assembler Format**
  
  LDUH @Rj, Ri

- **Operation**
  
  \( \text{extu(Rj)} \rightarrow Ri \)

- **Flag Change**
  
  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  - & - & - & - \\
  \end{array}
  \]

  N, Z, V, C: Unchanged.

- **Classification**
  
  Memory Load instruction, Instruction with delay slot

- **Execution Cycles**
  
  \( b \) cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rj</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ri</td>
</tr>
</tbody>
</table>
 Execution Example

LDUH @R2, R3 ; Bit pattern of the instruction: 0000 0101 0010 0011

Before execution

After execution
7.116 LDUH (Load Halfword Data in Memory to Register)

Extends with zeros the half-word data at memory address R13 + Rj, loads to Ri.

- **Assembler Format**
  
  LDUH @(R13, Rj), Ri

- **Operation**
  
  extu((R13+Rj)) → Ri

- **Flag Change**
  
  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  - & - & - & - \\
  \end{array}
  \]

  N, Z, V, C: Unchanged.

- **Classification**
  
  Memory Load instruction, Instruction with delay slot

- **Execution Cycles**
  
  b cycle

- **Instruction Format**
  
  \[
  \begin{array}{ccccccccc}
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & Rj & Ri \\
  \end{array}
  \]
 execution Example

LDUH @(R13, R2), R3  ; Bit pattern of the instruction: 0000 0001 0010 0011

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>R3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
| R13 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>R3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
| R13 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8

Memory

Before execution

After execution
7.117 LDUH (Load Halfword Data in Memory to Register)

Extends with zeros the half-word data at memory address R14 + o8 \times 2, loads to Ri. The value o8 is a signed calculation. The value of o8 \times 2 is specified in disp9.

- **Assembler Format**
  
  LDUH @(R14, disp9), Ri

- **Operation**
  
  \[ \text{extu}((R14+o8 \times 2)) \rightarrow \text{Ri} \]

- **Flag Change**
  
  \[ \begin{array}{cccc}
  N & Z & V & C \\
  - & - & - & - \\
  \end{array} \]

  N, Z, V, C: Unchanged.

- **Classification**
  
  Memory Load instruction, Instruction with delay slot

- **Execution Cycles**
  
  b cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>o8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ri</td>
</tr>
</tbody>
</table>
7.117 Execution Example

LDUH @(R14,2), R3 ; Bit pattern of the instruction: 0100 0000 0001 0011

Before execution After execution

<table>
<thead>
<tr>
<th>R3</th>
<th>x x x x x x x x</th>
<th>R3</th>
<th>0 0 0 0 4 3 2 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>R14</td>
<td>1 2 3 4 5 6 7 8</td>
<td>R14</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12345678</th>
<th>Memory</th>
<th>12345678</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234567A</td>
<td>4 3 2 1</td>
<td>1234567A</td>
<td>4 3 2 1</td>
</tr>
</tbody>
</table>
FR81 Family

7.118 LDUH (Load Halfword Data in Memory to Register)

Loads the half word data at memory address BP+u16 × 2 to Ri. Unsigned u16 value is calculated. The value in u16 × 2 is specified as udisp17.

- **Assembler Format**
  
  LD @(BP, udisp17), Ri
  
- **Operation**
  
  (BP+u16 × 2) → Ri
  
- **Flag Change**
  
  N, Z, V, C: Unchanged.
  
- **Classification**
  
  Memory Load instruction, FR81 family
  
- **Execution Cycles**
  
  a cycle
  
- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0</td>
</tr>
<tr>
<td>(n+2)</td>
<td>u16</td>
</tr>
</tbody>
</table>
  
- **EIT Occurrence and Detection**

  A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected.
7.119 LEAVE (Leave Function)

This instruction is used for stack frame release processing for high level languages.

• Assembler Format
  LEAVE

• Operation
  R14+4 → R15
  (R15-4) → R14

• Flag Change
  N, Z, V, C: Unchanged.

• Classification
  Other instructions, Instruction with delay slot

• Execution Cycles
  b cycle

• Instruction Format
  MSB
  | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | LSB

### Execution Example

**LEAVE** ; Bit pattern of the instruction: 1001 1111 1001 0000

<table>
<thead>
<tr>
<th>R14</th>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 F F F F F 4</td>
<td>7 F F F F F 4</td>
<td>8 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>7 F F F F F E C</td>
<td>7 F F F F F E C</td>
<td>7 F F F F F 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FFFFFFC</td>
<td>7FFFFFFC</td>
</tr>
<tr>
<td>7FFFFFF0</td>
<td>7FFFFFF0</td>
</tr>
<tr>
<td>7FFFFFF4</td>
<td>7FFFFFF4</td>
</tr>
<tr>
<td>7FFFFFF8</td>
<td>7FFFFFF8</td>
</tr>
<tr>
<td>7FFFFFFFC</td>
<td>7FFFFFFFC</td>
</tr>
<tr>
<td>80000000</td>
<td>80000000</td>
</tr>
</tbody>
</table>

Before execution | After execution
7.120  LSL (Logical Shift to the Left Direction)

Makes a logical left shift of the word data in Ri by Rj bits, stores the result to Ri. Only the lower 5 bits of Rj, which designates the size of the shift, are valid and the shift range is 0 to 31 bits.

- Assembler Format
  
  LSL    Rj, Ri

- Operation
  
  Ri << Rj → Ri

- Flag Change
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>C</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V: Unchanged.
  C: Holds the bit value shifted last. Cleared when the shift amount is zero.

- Classification
  
  Shift instruction, Instruction with delay slot

- Execution Cycles
  
  1 cycle

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
<th></th>
<th></th>
<th></th>
<th>Rj</th>
<th></th>
<th></th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
FR81 Family

Execution Example

```
LSL R2, R3 ; Bit pattern of the instruction: 1011 0110 0010 0011
```

Before execution

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
<th>CCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 8</td>
<td>F F F F F F F</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>F F F F F F F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
<th>CCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 8</td>
<td>F F F F F 0 0</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>F F F F F 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
7.121 LSL (Logical Shift to the Left Direction)

Makes a logical left shift of the word data in Ri by u4 bits, stores the result to Ri.

- **Assembler Format**
  
  LSL #u4, Ri

- **Operation**
  
  Ri << u4 → Ri

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>C</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V: Unchanged.
  C: Holds the bit value shifted last. Cleared when the shift amount is zero.

- **Classification**
  
  Shift instruction, Instruction with delay slot

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u4</td>
<td>Ri</td>
</tr>
</tbody>
</table>
FR81 Family

- Execution Example

```assembly
LSL #8, R3 ; Bit pattern of the instruction: 1011 0100 1000 0011
```

Before execution

<table>
<thead>
<tr>
<th>R3</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R3</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCR</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.122  LSL2 (Logical Shift to the Left Direction)

Makes a logical left shift of the word data in Ri by u4+16 bits, stores the result to Ri.

- Assembler Format
  
  LSL2 #u4, Ri

- Operation
  
  Ri << {u4+16} → Ri

- Flag Change
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>C</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V: Unchanged.
  C: Holds the bit value shifted last.

- Classification
  
  Shift instruction, Instruction with delay slot

- Execution Cycles
  
  1 cycle

- Instruction Format
  
  | MSB | | | | | | | LSB |
  |-----| | | | | | |-----|
  | 1   | 0 | 1 | 1 | 0 | 1 | 0 | 1 | u4 | Ri |
FR81 Family

- Execution Example

    LSL2 #8, R3 ; Bit pattern of the instruction: 1011 0101 1000 0011

    Before execution
    R3
    | F F F F F F F F |
    | N Z V C       |
    | 0 0 0 0       |

    After execution
    R3
    | F F 0 0 0 0 0 0 |
    | N Z V C       |
    | 1 0 0 1       |

Before execution

After execution
7.123 LSR (Logical Shift to the Right Direction)

Makes a logical right shift of the word data in Ri by Rj bits, stores the result to Ri. Only the lower 5 bits of Rj, which designates the size of the shift, are valid and the shift range is 0 to 31 bits.

- **Assembler Format**
  
  LSR  Rj, Ri

- **Operation**
  
  Ri >> Rj → Ri

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V: Unchanged.
  C: Holds the bit value shifted last. Cleared when the shift amount is zero.

- **Classification**

  Shift instruction, Instruction with delay slot

- **Execution Cycles**

  1 cycle

- **Instruction Format**

  MSB
  
<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>Rj</th>
<th>Ri</th>
</tr>
</thead>
</table>
FR81 Family

- Execution Example

    LSR  R2, R3 ; Bit pattern of the instruction: 1011 0010 0010 0011

<table>
<thead>
<tr>
<th>R2</th>
<th>0 0 0 0 0 0 0 8</th>
<th>R2</th>
<th>0 0 0 0 0 0 0 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>F F F F F F F F</td>
<td>R3</td>
<td>0 0 F F F F F F</td>
</tr>
<tr>
<td>NZVC</td>
<td>0 0 0 0</td>
<td>NZVC</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
<td>CCR</td>
<td>0 0 0 1</td>
</tr>
</tbody>
</table>

Before execution | After execution
7.124  LSR (Logical Shift to the Right Direction)

Makes a logical left shift of the word data in Ri by u4 bits, stores the result to Ri.

- **Assembler Format**
  
  LSR  #u4, Ri

- **Operation**
  
  Ri >> u4 → Ri

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>C</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  
  Z: Set when the operation result is zero, cleared otherwise.
  
  V: Unchanged.
  
  C: Holds the bit value shifted last. Cleared when the shift amount is zero.

- **Classification**
  
  Shift instruction, Instruction with delay slot

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u4</td>
<td>Ri</td>
</tr>
</tbody>
</table>
**FR81 Family**

* Execution Example

```
LSR  #8, R3 ; Bit pattern of the instruction: 1011 0000 1000 0011
```

<table>
<thead>
<tr>
<th>R3</th>
<th>00FF FFFFFFFF FFFF</th>
<th>R3</th>
<th>00 F F F F F F F F</th>
</tr>
</thead>
<tbody>
<tr>
<td>NZVC</td>
<td>0 0 0 0</td>
<td>NZVC</td>
<td>0 0 0 1</td>
</tr>
</tbody>
</table>

Before execution | After execution
7.125 LSR2 (Logical Shift to the Right Direction)

Makes a logical left shift of the word data in Ri by u4+16 bits, stores the result to Ri.

- **Assembler Format**
  
  LSR2 #u4, Ri

- **Operation**
  
  Ri >> \{u4+16\} → Ri

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>C</td>
</tr>
</tbody>
</table>

  N: Cleared.
  Z: Set when the operation result is zero, cleared otherwise.
  V: Unchanged.
  C: Holds the bit value shifted last.

- **Classification**

  Shift instruction, Instruction with delay slot

- **Execution Cycles**

  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>u4</td>
<td>Ri</td>
</tr>
</tbody>
</table>
FR81 Family

Execution Example

LSR2  #8, R3 ; Bit pattern of the instruction: 1011 0001 1000 0011

Before execution

R3: F F F F F F F F

CCR: 0 0 0 0

After execution

R3: 0 0 0 0 0 0 F F

CCR: 0 0 0 1

Before execution

After execution
7.126 MOV (Move Word Data in Source Register to Destination Register)

Moves the word data in Rj to Ri.

- **Assembler Format**
  
  MOV Rj, Ri

- **Operation**
  
  Rj → Ri

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**

  Inter-register transfer instruction, Instruction with delay slot

- **Execution Cycles**

  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 1 0 1 1</td>
<td>Rj</td>
</tr>
<tr>
<td></td>
<td>Ri</td>
</tr>
</tbody>
</table>
### Execution Example

```assembly
MOV R2, R3  ; Bit pattern of the instruction: 1000 1011 0010 0011
```

<table>
<thead>
<tr>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2 8 7 6 5 4 3 2 1</td>
<td>R2 8 7 6 5 4 3 2 1</td>
</tr>
<tr>
<td>R3 x x x x x x x x</td>
<td>R3 8 7 6 5 4 3 2 1</td>
</tr>
</tbody>
</table>

Before execution |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R2 8 7 6 5 4 3 2 1</td>
</tr>
<tr>
<td>R3 x x x x x x x</td>
</tr>
</tbody>
</table>
7.127 MOV (Move Word Data in Source Register to Destination Register)

Moves the word data in dedicated register Rs to general-purpose register Ri.

- **Assembler Format**
  
  \[
  \text{MOV} \quad \text{Rs}, \text{Ri}
  \]

- **Operation**
  
  \[
  \text{Rs} \rightarrow \text{Ri}
  \]

  If the number of a non-existent dedicated register is given as Rs, undefined data will be transferred.

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Inter-Register Transfer instruction, Instruction with delay slot

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Rs</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Ri</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
FR81 Family

● Execution Example

MOV MDL, R3 ; Bit pattern of the instruction: 1011 0111 0101 0011

<table>
<thead>
<tr>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>MDL</td>
</tr>
<tr>
<td>x x x x</td>
<td>8 7 6 5 4 3 2 1</td>
</tr>
<tr>
<td>8 7 6 5 4 3 2 1</td>
<td>MDL</td>
</tr>
</tbody>
</table>

Before execution After execution
7.128 MOV (Move Word Data in Program Status Register to Destination Register)

Moves the word data in the program status (PS) to general-purpose register Ri.

- Assembler Format
  
  MOV PS, Ri

- Operation
  
  PS → Ri

- Flag Change
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification
  
  Inter-Register Transfer instruction, Instruction with delay slot

- Execution Cycles
  
  1 cycle

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 0 1 1 1 0 0 0 1</td>
<td>Ri</td>
</tr>
</tbody>
</table>
FR81 Family

Execution Example

MOV  PS, R3  ; Bit pattern of the instruction: 0001 0111 0001 0011

Before execution                      After execution

R3: x x x x x x x x  R3: F F F 8 F 8 C 0
PS: F F F 8 F 8 C 0  PS: F F F 8 F 8 C 0
7.129 MOV (Move Word Data in Source Register to Destination Register)

Moves the word data of general-purpose register Ri to dedicated register Rs.

- Assembler Format
  
  MOV   Ri, Rs

- Operation
  
  Ri → Rs

  If TBR, SSP, or ESR is specified in user mode or if a number without a dedicated register is specified for "RS", it generates an invalid instruction exception (system-only register access).

  There is no restriction in privilege mode. If the number of a non-existent register is given as parameter Rs in privilege mode, the read value Ri will be ignored.

- Flag Change
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification
  
  Inter-Register Transfer instruction, Instruction with delay slot, FR81 updating

- Execution Cycles
  
  1 cycle

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 0 1 1 Rs Ri</td>
</tr>
<tr>
<td>LSBJ</td>
</tr>
</tbody>
</table>
FR81 Family

● EIT Occurrence and Detection

User mode:
  An invalid instruction exception (system-only register access) is generated and an interrupt is detected.

Privilege mode:
  An interrupt is detected.

● Execution Example

MOV R3, MDL ; Bit pattern of the instruction: 1011 0101 0011

<table>
<thead>
<tr>
<th>R3</th>
<th>MDL</th>
<th>Before execution</th>
<th>R3</th>
<th>MDL</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 7 6 5 4 3 2 1</td>
<td>x x x x x x x</td>
<td></td>
<td>8 7 6 5 4 3 2 1</td>
<td>8 7 6 5 4 3 2 1</td>
<td></td>
</tr>
</tbody>
</table>

Before execution After execution
7.130 MOV (Move Word Data in Source Register to Program Status Register)

Stores the word data of general-purpose register Ri to program status (PS).

- **Assembler Format**
  
  MOV Ri, PS

- **Operation**
  
  Ri → PS

The contents of system status register (SSR) cannot be changed by this instruction regardless of the selected operation mode. If this instruction is executed in user mode, only the D1, D0, N, Z, V, and C flags can be changed. The other flag values are not updated. Any bit other than SSR can be changed in privilege mode.

At the time this instruction is executed, if the value of the interrupt level mask register (ILM) is in the range 16 to 31, only new ILM settings between 16 and 31 can be entered. If data in the range 0 to 15 is loaded from Ri, the value +16 is transferred to the ILM. If the original ILM value is in the range 0 to 15, then any value from 0 to 31 can be transferred to the ILM.

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

N, Z, V, C: Data from Ri is transferred.

- **Classification**
  
  Inter-Register Transfer instruction, Instruction with delay slot, FR81 updating

- **Execution Cycles**
  
  1 cycle
FR81 Family

- Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Ri</td>
<td></td>
</tr>
</tbody>
</table>

- EIT Occurrence and Detection

An interrupt is detected.

- Execution Example

MOV R3, PS ; Bit pattern of the instruction: 0000 0111 0001 0011

R3 | F F F 3 F 8 D 5 |
PS | x x x x x x x x |

Before execution

R3 | F F F 3 F 8 D 5 |
PS | F F F 3 F 8 D 5 |

After execution
7.131 MOV (Move Word Data in General Purpose Register to Floating Point Register)

The value in Rj is transferred to FRi.

- **Assembler Format**
  
  MOV Rj, FRi

- **Operation**
  
  Rj → FRi

- **Classification**
  
  Single-precision floating point instruction, FR81 family

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0</td>
</tr>
<tr>
<td>(n+2)</td>
<td>-</td>
</tr>
</tbody>
</table>

- **EIT Occurrence and Detection**
  
  An invalid instruction exception (FPU absence error) or an interrupt is detected.
7.132 MOV (Move Word Data in Floating Point Register to General Purpose Register)

The value in FRi is transferred to Rj.

- Assembler Format
  
  MOV FRi, Rj

- Operation
  
  FRi → Rj

- Classification
  
  Single-precision floating point instruction, FR81 family

- Execution Cycles
  
  1 cycle

- Instruction Format
  
  \[
  \begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c}
  \hline
  \text{MSB} & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & \text{Rj} \\
  \text{(n+0)} & \hline
  \text{(n+2)} & - & - & - & - & \text{FRi} \\
  \hline
  \end{array}
  \]

- EIT Occurrence and Detection
  
  An invalid instruction exception (FPU absence error) or an interrupt is detected.
7.133  MUL (Multiply Word Data)

Multiplies the word data in Rj by the word data in Ri as signed numbers, and stores the resulting signed 64-bit data with the higher word in the multiplication/division register (MDH), and the lower word in the multiplication/division register (MDL).

- **Assembler Format**
  
  MUL   Rj, Ri

- **Operation**
  
  Ri × Rj → MDH, MDL

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>-</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the "MDL" of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V: Cleared when the operation result is in the range -2147483648 to 2147483647, cleared otherwise.
  C: Unchanged.

- **Classification**

  Multiply/Divide Instruction

- **Execution Cycles**

  5 cycles

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Rj</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Ri</td>
<td></td>
</tr>
</tbody>
</table>
### Execution Example

MUL R2, R3  ; Bit pattern of the instruction: 1010 1111 0010 0011

<table>
<thead>
<tr>
<th></th>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>0 0 0 0 0 0 0 2</td>
<td>0 0 0 0 0 0 0 2</td>
</tr>
<tr>
<td>R3</td>
<td>8 0 0 0 0 0 0 1</td>
<td>8 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>MDH</td>
<td>x x x x x x x x</td>
<td>F F F F F F F F</td>
</tr>
<tr>
<td>MDL</td>
<td>x x x x x x x x</td>
<td>0 0 0 0 0 0 0 2</td>
</tr>
<tr>
<td>N Z V C</td>
<td>0 0 0 0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
<td>0 0 1 0</td>
</tr>
</tbody>
</table>
7.134 MULH (Multiply Halfword Data)

Multiplies the half-word data in the lower 16 bits of Rj by the half-word data in the lower 16 bits of Ri as signed numbers, and stores the resulting signed 32-bit data in the multiplication/division register (MDL). The multiplication/division register (MDH) is undefined.

- **Assembler Format**
  
  MULH Rj, Ri

- **Operation**
  
  Ri × Rj → MDL

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the MDL of the operation result is "1", cleared when the MSB is "0".
  
  Z: Set when MDL of the operation result is zero, cleared otherwise.
  
  V, C: Unchanged.

- **Classification**
  
  Multiply/Divide Instruction

- **Execution Cycles**
  
  3 cycles

- **Instruction Format**
  
  MSB
  
<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>Rj</th>
<th>Ri</th>
</tr>
</thead>
</table>
FR81 Family

- Execution Example

MULH R2, R3 ; Bit pattern of the instruction: 1011 1111 0010 0011

Before execution After execution

<table>
<thead>
<tr>
<th>R2</th>
<th>F E D C B A 9 8</th>
<th>R2</th>
<th>F E D C B A 9 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>0 1 2 3 4 5 6 7</td>
<td>R3</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>MDH</td>
<td>x x x x x x x</td>
<td>MDH</td>
<td>x x x x x x x</td>
</tr>
<tr>
<td>MDL</td>
<td>x x x x x x x</td>
<td>MDL</td>
<td>E D 2 F 0 B 2 8</td>
</tr>
<tr>
<td>CCR</td>
<td>0 0 0 0</td>
<td>CCR</td>
<td>1 0 0 0</td>
</tr>
</tbody>
</table>

Before execution After execution
7.135 MULU (Multiply Unsigned Word Data)

Multiplies the word data in Rj by the word data in Ri as unsigned numbers and stores the resulting unsigned 64-bit data with the higher word in the multiplication/division register (MDH), and the lower word in the multiplication/division register (MDL).

- **Assembler Format**
  
  MULU  Rj, Ri

- **Operation**
  
  \[ R_i \times R_j \rightarrow MDH, MDL \]

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>-</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the MDL of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the MDL of the operation result is zero, cleared otherwise.
  V: Cleared when the operation result is in the range 0 to 4294967295, set otherwise.
  C: Unchanged.

- **Classification**

  Multiply/Divide Instruction

- **Execution Cycles**

  5 cycles

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Rj</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ri</td>
</tr>
</tbody>
</table>
**Execution Example**

MULU R2, R3 ; Bit pattern of the instruction: 1010 1011 0010 0011

<table>
<thead>
<tr>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2 00000002</td>
<td>R2 00000002</td>
</tr>
<tr>
<td>R3 80000001</td>
<td>R3 80000001</td>
</tr>
<tr>
<td>MDH xxxxxxx</td>
<td>MDH 00000001</td>
</tr>
<tr>
<td>MDL xxxxxxx</td>
<td>MDL 00000002</td>
</tr>
<tr>
<td>NZVC</td>
<td>NZVC</td>
</tr>
<tr>
<td>CCR 0000</td>
<td>CCR 0010</td>
</tr>
</tbody>
</table>
7.136 MULUH (Multiply Unsigned Halfword Data)

Multiplies the half-word data in the lower 16 bits of Rj by the half-word data in the lower 16 bits of Ri as unsigned numbers, and stores the resulting unsigned 32-bit data in the multiplication/division register (MDL). The multiplication/division register (MDH) is undefined.

- **Assembler Format**
  
  MULUH Rj, Ri

- **Operation**
  
  Ri × Rj → MDL

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the MDL of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the MDL of the operation result is zero, cleared otherwise.
  V, C: Unchanged.

- **Classification**
  
  Multiply/Divide Instruction

- **Execution Cycles**
  
  3 cycles

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Rj</td>
</tr>
<tr>
<td></td>
<td>Ri</td>
</tr>
</tbody>
</table>
FR81 Family

Execution Example

MULUH R2, R3  ; Bit pattern of the instruction: 1011 1011 0010 0011

Before execution

R2 | F E D C B A 9 8  
R3 | 0 1 2 3 4 5 6 7  
MDH | x x x x x x x x  
MDL | x x x x x x x x  
CCR | 0 0 0 0

After execution

R2 | F E D C B A 9 8  
R3 | 0 1 2 3 4 5 6 7  
MDH | x x x x x x x x  
MDL | 3 2 9 6 0 B 2 8  
CCR | 0 0 0 0
7.137 NOP (No Operation)

This instruction performs no operation.

- **Assembler Format**
  
  ```
  NOP
  ```

- **Operation**
  
  No operation

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**

  Other instructions, Instruction with delay slot

- **Execution Cycles**

  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>1 0 0 1 1 1 1 1 1 1 0 1 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td></td>
</tr>
</tbody>
</table>


Execution Example

NOP ; Bit pattern of the instruction: 1001 1111 1010 0000

Before execution | After execution
7.138 OR (Or Word Data of Source Register to Data in Memory)

Takes the logical OR of the word data at memory address Ri and the word data in Rj, stores the results to the memory address corresponding to Ri.

- Assembler Format
  
  \[ \text{OR Rj, @Ri} \]

- Operation
  
  \[(Ri) | Rj \rightarrow (Ri)\]

- Flag Change
  
  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  C & C & - & - \\
  \end{array}
  \]

  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V, C: Unchanged.

- Classification
  
  Logical Calculation instruction, Read/Modify/Write type instruction

- Execution Cycles
  
  1+2a cycles

- Instruction Format
  
  \[
  \begin{array}{cccccc}
  \text{MSB} & & & & & \\
  1 & 0 & 0 & 1 & 0 & 0 \end{array} \quad \begin{array}{c}
  \text{Rj}
  \end{array} \quad \begin{array}{c}
  \text{Ri}
  \end{array}
  \]
FR81 Family

Execution Example

OR R2, @R3 ; Bit pattern of the instruction: 1001 0100 0010 0011

<table>
<thead>
<tr>
<th>R2</th>
<th>1 1 1 1 0 0 0 0</th>
<th>R2</th>
<th>1 1 1 1 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>1 2 3 4 5 6 7 8</td>
<td>R3</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

Memory

12345678 | 1 0 1 0 1 0 1 0 |
1234567C | 1 1 1 1 1 0 1 0 |

NZVC

| CCR | 0 0 0 0 | CCR | 0 0 0 0 |

Before execution

After execution
7.139 OR (Or Word Data of Source Register to Destination Register)

Takes the logical OR of the word data in Ri and the word data in Rj, stores the results to Ri.

- **Assembler Format**

  
  OR   Rj, Ri

- **Operation**

  
  Ri | Rj → Ri

- **Flag Change**

  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is zero, cleared otherwise.
  V, C: Unchanged.

- **Classification**

  Logical Calculation instruction, Instruction with delay slot

- **Execution Cycles**

  1 cycle

- **Instruction Format**

  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Rj</td>
<td>Ri</td>
</tr>
</tbody>
</table>
FR81 Family

- Execution Example

```
OR  R2, R3 ; Bit pattern of the instruction: 1001 0010 0010 0011
```

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
<th>N Z V C</th>
<th>CCR</th>
<th>R2</th>
<th>R3</th>
<th>N Z V C</th>
<th>CCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 0000</td>
<td>1010 1010</td>
<td>1111 0000</td>
<td>1111 1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before execution     After execution
7.140 ORB (Or Byte Data of Source Register to Data in Memory)

Takes the logical OR of the byte data at memory address Ri and the byte data in Rj, stores the results to the memory address corresponding to Ri.

- Assembler Format
  
  ORB  Rj,@Ri

- Operation
  
  \[(Ri) \mid Rj \rightarrow (Ri)\]

- Flag Change
  
  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  \hline
  C & C & - & -
  \end{array}
  \]

  N: Set when the MSB(bit7) of the operation result is "1", cleared when the MSB(bit7) is "0".
  
  Z: Set when the operation result is zero, cleared otherwise.
  
  V, C: Unchanged.

- Classification
  
  Logical Calculation instruction, Read/Modify/Write type instruction

- Execution Cycles
  
  1+2a cycles

- Instruction Format
  
  \[
  \begin{array}{ccccccccc}
  \text{MSB} & & & & & & & & \\
  1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & \text{Rj} \\
  & & & & & & & & \\
  & & & & & & & & \text{Ri}
  \end{array}
  \]
FR81 Family

Execution Example

ORB R2, @R3 ; Bit pattern of the instruction: 1001 0110 0010 0011

<table>
<thead>
<tr>
<th>R2</th>
<th>0 0 0 0 0 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>12345678</th>
<th>N Z V C</th>
<th>CCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before execution</td>
<td>1 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>After execution</td>
<td>1 1</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

Memory | 12345678 | N Z V C | CCR |
--------|----------|--------|-----|
| Before execution | 1 0 | 0 0 0 0 | 0 0 0 0 |
| After execution   | 1 1 | 0 0 0 0 | 0 0 0 0 |
7.141 ORCCR (Or Condition Code Register and Immediate Data)

Takes the logical OR of the byte data in the condition code register (CCR) and the immediate data, and returns the results in to the CCR.

- **Assembler Format**
  
  ORCCR #u8

- **Operation**
  
  User mode:
  
  $\text{CCR} \mid (\text{u8} \& \text{CF}_H) \rightarrow \text{CCR}$
  
  Privilege mode:
  
  $\text{CCR} \mid \text{u8} \rightarrow \text{CCR}$

  In user mode, a request to rewrite the stack flag (S) or the interrupt enable flag (I) is ignored. The S and I flags can only be changed in privilege mode.

- **Flag Change**
  
<table>
<thead>
<tr>
<th>S</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

  S, I, N, Z, V, C: Varies according to results of calculation.

- **Classification**
  
  Other instructions, Instruction with delay slot, FR81 updating

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 0 0 1 1</td>
<td>u8</td>
</tr>
</tbody>
</table>
FR81 Family

- **EIT Occurrence and Detection**
  
  An interrupt is detected (the value of I flag after instruction execution is used).

- **Execution Example**

  
  ORCCR   #10H ; Bit pattern of the instruction: 1001 0011 0001 0000

  
  Before execution

<table>
<thead>
<tr>
<th>S</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCR</td>
<td>0 0 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  After execution

<table>
<thead>
<tr>
<th>S</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCR</td>
<td>0 1 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS
7.142

7.142 ORH (Or Halfword Data of Source Register to Data in Memory)

Takes the logical OR if the half-word data at memory address Ri and the half-word data in Rj, stores the results to the memory address corresponding to Ri.

- **Assembler Format**
  
  ORH  Rj,@Ri

- **Operation**
  
  \((Ri) \mid R_j \rightarrow (Ri)\)

- **Flag Change**
  
  \(\begin{array}{cccc}
  N & Z & V & C \\
  C & C & \_ & \_ \\
  \end{array}\)

  N: Set when the MSB(bit15) of the operation result is "1", cleared when the MSB is "0".

  Z: Set when the operation result is zero, cleared otherwise.

  V, C: Unchanged.

- **Classification**
  
  Logical Calculation instruction, Read/Modify/Write type instruction

- **Execution Cycles**
  
  1+2a cycles

- **Instruction Format**
  
  \[
  \begin{array}{cccccccc}
  & & & & & \text{Rj} & \text{Ri} \\
  1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
  \end{array}\]
FR81 Family

Execution Example

ORH R2,@R3 ; Bit pattern of the instruction: 1001 0101 0010 0011

| R2 | 0 0 0 0 1 1 0 0 |
| R3 | 1 2 3 4 5 6 7 8 |

Before execution

| Memory | 1 0 1 0 |
| 12345678 | |
| 1234567A | |

| N Z V C | 0 0 0 0 |
| CCR | |

After execution

| Memory | 1 1 1 0 |
| 12345678 | |
| 1234567A | |

| N Z V C | 0 0 0 0 |
| CCR | |
7.143 RET (Return from Subroutine)

This is a branching instruction without a delay slot. Branches to the address indicated by the return pointer (RP). Used for return from Subroutine.

- **Assembler Format**
  
  RET

- **Operation**
  
  RP → PC

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Non-Delayed Branching instruction

- **Execution Cycles**
  
  2 cycles

- **Instruction Format**
  
  MSB ──── LSB
  
  
  1 0 0 1 0 1 1 1 0 0 1 0 0 0 0 0
**FR81 Family**

- Execution Example

RET ; Bit pattern of the instruction: 1001 0111 0010 0000

<table>
<thead>
<tr>
<th>PC</th>
<th>RP</th>
<th>Before execution</th>
<th>PC</th>
<th>RP</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>F F 0 8 8 2 0</td>
<td>8 0 0 0 A E 8 6</td>
<td></td>
<td>8 0 0 0 A E 8 6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before execution

After execution
7.144 RET:D (Return from Subroutine)

This is a branching instruction with a delay slot. Branches to the address indicated by the return pointer (RP). Used for return from Subroutine.

- Assembler Format
  
  RET:D

- Operation
  
  RP → PC

- Flag Change
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification
  
  Delayed Branching instruction

- Execution Cycles
  
  1 cycle

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 1 1 1 1 0 0 1 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
Execution Example

```
RET: D ; Bit pattern of the instruction: 1001 1111 0010 0000
MOV  R0, R1 ; Instruction placed in delay slot
```

Before execution of RET instruction

<table>
<thead>
<tr>
<th>R0</th>
<th>R0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00112233</td>
<td>00112233</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxxxxxx</td>
<td>xxxxxxxx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFF08820</td>
<td>8000A86</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RP</th>
<th>RP</th>
</tr>
</thead>
<tbody>
<tr>
<td>8000A86</td>
<td>8000A86</td>
</tr>
</tbody>
</table>

After branching

```
```

The instruction placed in delay slot will be executed before execution of the branch destination instruction. The value of R1 above will vary according to the specifications of the MOV instruction placed in the delay slot.
7.145  RETI (Return from Interrupt)

Loads data from the stack indicated by system stack pointer (SSP), to the program counter (PC) and program status (PS), and retakes control from the EIT operation handler.

- **Assembler Format**
  
  RETI

- **Operation**
  
  - Normal operation state
    
    (SSP) → PC
    SSP+4 → SSP
    (SSP) → PS
    SSP+4 → SSP
  
  - Debug state
    
    PC save register (PCSR) → PC
    PS save register (PSSR) → PS
    (PC) instruction execution

This is a privilege instruction, which is only available in privilege mode. If this instruction is executed in user mode, it generates an invalid instruction exception (privilege instruction execution).

Operation varies depending on whether this instruction is executed in the normal operation state or debug state. If this instruction is executed in the debug state, the DSU register is used instead of a stack, and the acceptance of interrupts is withheld until the next instruction execution is completed. This, therefore, executes one instruction necessarily after the RETI instruction was executed in the debug state.

At the time this instruction is executed, if the value of the interrupt level mask register (ILM) is in the range 16 to 31, only new ILM settings between 16 and 31 can be entered. If data in the range 0 to 15 is loaded in memory, the value 16 will be added to that data before being transferred to the ILM. If the original ILM value is in the range 0 to 15, then any value between 0 and 31 can be transferred to the ILM.

- **Flag Change**
  
<table>
<thead>
<tr>
<th>S</th>
<th>I</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

D2, D1, S, I, N, Z, V, C: Change according to the values retrieved from the stack.
CHAPTER 7 DETAILED EXECUTION INSTRUCTIONS

FR81 Family

● Classification
  Non-Delayed Branching instruction, FR81 updating

● Execution Cycles
  1+2b cycles

● Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

● EIT Occurrence and Detection

User mode:
  An invalid instruction exception (privilege instruction execution) is generated.

Privilege mode:
  A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected. The interrupt level is judged using the value returned from the stack.

Debug state:
  EIT is not accepted.
FR81 Family

CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS
7.145

Execution Example

RETI ; Bit pattern of the instruction: 1001 0111 0011 0000

<table>
<thead>
<tr>
<th>R15</th>
<th>7 F F F F F F F 8</th>
<th>R15</th>
<th>4 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSP</td>
<td>7 F F F F F F F 8</td>
<td>SSP</td>
<td>8 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>USP</td>
<td>4 0 0 0 0 0 0 0</td>
<td>USP</td>
<td>4 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>PC</td>
<td>F F 0 0 9 0 B C</td>
<td>PC</td>
<td>8 0 8 8 8 0 8 8</td>
</tr>
<tr>
<td>PS</td>
<td>F F F 0 F 8 D 4</td>
<td>PS</td>
<td>F F F 3 F 8 F 1</td>
</tr>
<tr>
<td>ILM</td>
<td>1 0 0 0 0</td>
<td>ILM</td>
<td>1 0 0 1 1</td>
</tr>
<tr>
<td>CCR</td>
<td>0 1 0 1 0 0</td>
<td>CCR</td>
<td>1 1 0 0 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>7 F F F F F F F 8</th>
<th>Memory</th>
<th>7 F F F F F F F 8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 0 8 8 8 0 8 8</td>
<td></td>
<td>8 0 8 8 8 0 8 8</td>
</tr>
<tr>
<td></td>
<td>7 F F F F F F F C</td>
<td></td>
<td>7 F F F F F F F C</td>
</tr>
<tr>
<td></td>
<td>F F F 3 F 8 F 1</td>
<td></td>
<td>F F F 3 F 8 F 1</td>
</tr>
<tr>
<td></td>
<td>x x x x x x x x</td>
<td></td>
<td>x x x x x x x x</td>
</tr>
</tbody>
</table>

Before execution | After execution
### 7.146 SRCH0 (Search First Zero bit position distance From MSB)

This is a "0" search instruction used for bit searching. Takes a comparison of word data in Ri from MSB (bit31) and "0", stores the distance in Ri from the first "0" that is found and bit MSB (bit31).

#### Assembler Format

SRCH0 Ri

#### Operation

search_zero(Ri) → Ri

If "0" bit is not found (in case all word data of Ri is "1" bit), 32 is stored in Ri. In case MSB(bit31) is "0", zero is stored in Ri and 31 is stored in Ri when LSB (bit0) is "0" and other bits are "1".

The Ri bit pattern before execution of instruction its relation with the values stored in Ri is shown in Table 7.146-1.

#### Table 7.146-1 Input pattern of SRCH0 instruction and its results

<table>
<thead>
<tr>
<th>Input (Ri bit pattern before execution of instruction)</th>
<th>Result</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111111_11111111_11111111_11111111</td>
<td>32</td>
<td>Not found</td>
</tr>
<tr>
<td>0xxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx</td>
<td>0</td>
<td>MSB(bit31) was &quot;1&quot;</td>
</tr>
<tr>
<td>10xxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>110xxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1110xxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111111_11111111_11111111_1111110xxx</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>11111111_11111111_11111111_11110xx</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>11111111_11111111_11111111_111110x</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>11111111_11111111_11111111_1111110</td>
<td>31</td>
<td>Only LSB(bit0) was &quot;0&quot;</td>
</tr>
</tbody>
</table>

#### Flag Change

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

N, Z, V, C: Unchanged.
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

7.146

FR81 Family

● Classification

Bit Search instruction, Instruction with delay slot

● Execution Cycles

1 cycle

● Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 0 1 1 1 1 1 0 0</td>
<td>Ri</td>
</tr>
</tbody>
</table>

● Execution Example

SRCH0  R2 ; Bit pattern of the instruction: 1001 0111 1100 0010

Before execution

R2  F C 3 4 5 6 7 8

After execution

R2  0 0 0 0 0 0 0 6
FR81 Family

7.147 SRCH1 (Search First One bit position distance From MSB)

This is a "1" search instruction used for bit searching. Takes a comparison of word data in Ri from MSB (bit31) and "1", stores the distance in Ri from the first "1" that is found and bit MSB(bit31).

- Assembler Format

```
SRCH1 Ri
```

- Operation

```
search_one(Ri) → Ri
```

If "1" bit is not found (in case all word data of Ri is "0" bit), 32 is stored in Ri. In case MSB(bit31) is "1", zero is stored in Ri and 31 is stored in Ri when LSB (bit0) is "1" and other bits are "0".

The Ri bit pattern before execution of instruction its relation with the values stored in Ri is shown in Table 7.147-1.

<table>
<thead>
<tr>
<th>Input (Ri bit pattern before execution of the instruction)</th>
<th>Results</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000_00000000_00000000_00000000</td>
<td>32</td>
<td>Not found</td>
</tr>
<tr>
<td>1xxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx</td>
<td>0</td>
<td>MSB(bit31) was &quot;0&quot;</td>
</tr>
<tr>
<td>01xxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>001xxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0001xxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000_00000000_00000000_00000000_00001xxx</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>00000000_00000000_00000000_0000001xx</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>00000000_00000000_00000000_00000001x</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>00000000_00000000_00000000_00000000_00000001</td>
<td>31</td>
<td>Only LSB(bit0) was &quot;0&quot;</td>
</tr>
</tbody>
</table>

- Flag Change

```
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

N, Z, V, C: Unchanged.
Classification

Bit Search instruction, Instruction with delay slot

Execution Cycles

1 cycle

Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Ri</td>
</tr>
</tbody>
</table>

Execution Example

SRCH0 R2 ; Bit pattern of the instruction: 1001 0111 1101 0010

Before execution

R2 0 0 3 4 5 6 7 8

After execution

R2 0 0 0 0 0 0 0 A
7.148  SRCHC (Search First bit value change position distance From MSB)

This is a Change point search instruction used for bit searching. Takes a comparison of data in Ri with MSB (bit31), stores in Ri the distance from the first varying bit value that is found and bit MSB (bit31).

● Assembler Format

SRCHC Ri

● Operation

search_change(Ri) → Ri

If the values of all bits are the same, 32 is stored in Ri. In case the values of MSB (bit31) the adjacent bit30 is different, 1 is stored in Ri and 31 is stored in Ri when only the value of LSB (bit0) is different.

The Ri bit pattern before execution of instruction its relation with the values stored in Ri is shown in Table 7.148-1.

Table 7.148-1  Input bit pattern of SRCHC instruction and its results

<table>
<thead>
<tr>
<th>Input (Ri bit pattern before execution of instruction)</th>
<th>Results</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000_00000000_00000000_00000000_00000000_11111111_11111111_11111111_11111111</td>
<td>32</td>
<td>Not found</td>
</tr>
<tr>
<td>01xxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx</td>
<td>1</td>
<td>Difference between bit value of MSB(bit31) and bit30</td>
</tr>
<tr>
<td>001xxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0001xxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>00001xxx_xxxxxxxx_xxxxxxxx_xxxxxxxx</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>. . .</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000_00000000_00000000_00001xxx_11111111_11111111_11111111_11111111_11111111</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>00000000_00000000_00000000_000001xx_11111111_11111111_11111111_11111111_11111111</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>00000000_00000000_00000000_0000001x_11111111_11111111_11111111_11111111_11111111</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>00000000_00000000_00000000_00000001_11111111_11111111_11111111_11111111_11111111</td>
<td>31</td>
<td>Bit value of only LSB(bit0) is different</td>
</tr>
</tbody>
</table>

* The value of result (Ri) can not become 0.
● **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

N, Z, V, C: Unchanged.

● **Classification**

Bit Search instruction, Instruction with delay slot

● **Execution Cycles**

1 cycle

● **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 0 1 1 1 1 1 1 0</td>
<td>Ri</td>
</tr>
</tbody>
</table>

● **Execution Example**

SRCHC R2 ; Bit pattern of the instruction: 1001 0111 1110 0010

Before execution | After execution
--- | ---
R2 F F 3 4 5 6 7 8 | R2 0 0 0 0 0 0 0 8
7.149 ST (Store Word Data in Register to Memory)

Loads word data in Ri to memory address Rj.

- **Assembler Format**
  
  \[
  \text{ST} \quad \text{Ri,}@\text{Rj}
  \]

- **Operation**
  
  \[\text{Ri} \rightarrow (\text{Rj})\]

- **Flag Change**
  
  \[
  \begin{array}{cccc}
  \text{N} & \text{Z} & \text{V} & \text{C} \\
  \hline
  - & - & - & - \\
  \end{array}
  \]
  
  N, Z, V, C: Unchanged.

- **Classification**
  
  Memory Store instruction, Instruction with delay slot

- **Execution Cycles**
  
  a cycle

- **Instruction Format**
  
  \[
  \begin{array}{cccccc}
  0 & 0 & 0 & 1 & 0 & 1 \\
  \hline
  \text{Rj} & \text{Ri} \\
  \end{array}
  \]
Execution Example

ST R3,@R2 ; Bit pattern of the instruction: 0001 0100 0010 0011

Before execution  After execution

| R2 | 1 2 3 4 5 6 7 8 |
| R3 | 8 7 6 5 4 3 2 1 |

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
</tr>
</tbody>
</table>
7.150 ST (Store Word Data in Register to Memory)

Loads the word data in Ri to memory address R13 + Rj.

- **Assembler Format**
  
  \[ \text{ST} \ Ri, @(R13, Rj) \]

- **Operation**
  
  \[ Ri \rightarrow (R13+Rj) \]

- **Flag Change**
  
  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  - & - & - & - \\
  \end{array}
  \]

  N, Z, V, C: Unchanged.

- **Classification**
  
  Memory Store instruction, Instruction with delay slot

- **Execution Cycles**
  
  a cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rj</td>
<td></td>
</tr>
<tr>
<td>Ri</td>
<td></td>
</tr>
</tbody>
</table>
Execution Example

ST R3, @(R13, R2) ; Bit pattern of the instruction: 0001 0000 0010 0011

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
<th>R13</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 4</td>
<td>8 7 6 5 4 3 2 1</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

Before execution

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x x x x x</td>
</tr>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>1234567C</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 7 6 5 4 3 2 1</td>
</tr>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>1234567C</td>
</tr>
</tbody>
</table>
7.151 ST (Store Word Data in Register to Memory)

Loads the word data in Ri to memory address R14 + o8 \times 4. The value o8 is a signed calculation. The value of o8 \times 4 is specified in disp10.

- **Assembler Format**
  
  \[
  \text{ST} \ Ri, @(R14, \text{disp10})
  \]

- **Operation**
  
  \[
  Ri \rightarrow (R14+o8 \times 4)
  \]

- **Flag Change**
  
  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  \hline
  - & - & - & -
  \end{array}
  \]

  N, Z, V, C: Unchanged.

- **Classification**
  
  Memory Store instruction, Instruction with delay slot

- **Execution Cycles**
  
  a cycle

- **Instruction Format**

  \[
  \begin{array}{ccccccc}
  \text{MSB} & & & & & & \text{LSB} \\
  0 & 0 & 1 & 1 & o8 & & Ri
  \end{array}
  \]
FR81 Family

CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS
7.151

● Execution Example

ST  R3, @(R14,4) ; Bit pattern of the instruction: 0011 0000 0001 0011

Before execution

R3

8 7 6 5 4 3 2 1

R14

1 2 3 4 5 6 7 8

12345678

Memory

1234567C

xx x x xx

After execution

R3

8 7 6 5 4 3 2 1

R14

1 2 3 4 5 6 7 8

12345678

Memory

1234567C

8 7 6 5 4 3 2 1
7.152 ST (Store Word Data in Register to Memory)

Loads the word data in Ri to memory address R15 + u4 × 4. The value u4 is an unsigned calculation. The value of u4 × 4 is specified in udisp6.

- Assembler Format
  \[ \text{ST} \quad Ri, @(R15, \text{udisp6}) \]

- Operation
  \[ Ri \rightarrow (R15+u4 \times 4) \]

- Flag Change
  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  - & - & - & - \\
  \end{array}
  \]
  N, Z, V, C: Unchanged.

- Classification
  Memory Store instruction, Instruction with delay slot

- Execution Cycles
  a cycle

- Instruction Format
  \[
  \begin{array}{cccccc}
  \text{MSB} & & & & & \text{LSB} \\
  0 & 0 & 0 & 1 & 0 & 0 \quad 1 \quad 1 \quad u4 \quad Ri \\
  \end{array}
  \]
Execution Example

```
ST R3,@(R15,4) ; Bit pattern of the instruction: 0001 0011 0001 0011
```

Before execution

<table>
<thead>
<tr>
<th>R3</th>
<th>Memory</th>
<th>12345678</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 7 6 5 4 3 2 1</td>
<td>x x x x x x x x</td>
<td>1234567C</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R3</th>
<th>Memory</th>
<th>12345678</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 7 6 5 4 3 2 1</td>
<td>8 7 6 5 4 3 2 1</td>
<td>1234567C</td>
</tr>
</tbody>
</table>
Subtracts 4 from the value of R15, stores the word data in Ri to the memory address indicated by the new value of R15. If R15 is given as the parameter Ri, the data transfer will use the value of R15 before subtraction.

● Assembler Format

ST  Ri,@-R15

● Operation

R15 - 4 → R15
Ri → (R15)

● Flag Change

\[
\begin{array}{cccc}
N & Z & V & C \\
\hline
- & - & - & - \\
\end{array}
\]

N, Z, V, C: Unchanged.

● Classification

Memory Store instruction, Instruction with delay slot

● Execution Cycles

a cycle

● Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ri</td>
<td></td>
</tr>
</tbody>
</table>
Execution Example

ST R3, @-R15 ; Bit pattern of the instruction: 0001 0111 0000 0011

R3
8 7 6 5 4 3 2 1
R15
1 2 3 4 5 6 7 8

Memory
12345674 x x x x x x x
12345678

Before execution

R3
8 7 6 5 4 3 2 1
R15
1 2 3 4 5 6 7 4

Memory
12345674 8 7 6 5 4 3 2 1
12345678

After execution
 Loads the word data in Ri to memory address BP+u16 × 4. Unsigned u16 value is calculated. The value in u16 × 4 is specified as udisp18.

- **Assembler Format**
  
  \[
  \text{ST} \quad \text{Ri, @(BP, udisp18)}
  \]

- **Operation**
  
  \[
  \text{Ri} \rightarrow (\text{BP+u16} \times 4)
  \]

- **Flag Change**
  
  N, Z, V, C: Unchanged.

- **Classification**
  
  Memory Store instruction, FR81 family

- **Execution Cycles**

  a cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n+0)</td>
<td>0 0 0 1 0 1 1 1 0 1 0 0</td>
</tr>
<tr>
<td>(n+2)</td>
<td></td>
</tr>
</tbody>
</table>

- **EIT Occurrence and Detection**

  A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected.
7.155  ST (Store Word Data in Register to Memory)

Subtracts 4 from the value R15, stores the word data in dedicated register Rs to the memory address indicated by the new value of R15.

● Assembler Format
  
  ST  Rs, @-R15

● Operation
  
  R15 - 4 \rightarrow R15
  Rs \rightarrow (R15)

If the number of a non-existent dedicated register is specified in parameter Rs, the read value will be ignored.

● Flag Change

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N, Z, V, C: Unchanged.

● Classification

  Memory Store instruction, Instruction with delay slot

● Execution Cycles

  a cycle

● Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rs</td>
<td></td>
</tr>
</tbody>
</table>
Execution Example

ST MDH, @R15 ; Bit pattern of the instruction: 0001 0111 1000 0100

Before execution

12345670 | Memory |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x x x x x</td>
<td></td>
</tr>
</tbody>
</table>

12345674 | Memory |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x x x x</td>
<td></td>
</tr>
</tbody>
</table>

After execution

12345670 | Memory |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8 7 6 5 4 3 2 1</td>
<td></td>
</tr>
</tbody>
</table>

12345674 | Memory |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8 7 6 5 4 3 2 1</td>
<td></td>
</tr>
</tbody>
</table>
7.156 ST (Store Word Data in Program Status Register to Memory)

Subtracts 4 from the value of R15, stores the word data in the program status (PS) to the memory address indicated by the new value of R15.

- **Assembler Format**
  
  \[
  \text{ST} \quad \text{PS},@-\text{R15}
  \]

- **Operation**
  
  \[
  \begin{align*}
  \text{R15} - 4 & \rightarrow \text{R15} \\
  \text{PS} & \rightarrow (\text{R15})
  \end{align*}
  \]

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Memory Store instruction, Instruction with delay slot

- **Execution Cycles**
  
  a cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 0 1 1 1 1 1 0 0 1 0 0 0</td>
<td>0</td>
</tr>
</tbody>
</table>

  `NZVC`
FR81 Family

Execution Example

ST PS, @-R15 ; Bit pattern of the instruction: 0001 0111 1001 0000

Before execution

R15

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

PS

| F | F | F | 8 | F | 8 | C | 0 |

Memory

<table>
<thead>
<tr>
<th>12345670</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x x x x x</td>
</tr>
</tbody>
</table>

12345674

After execution

R15

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 4 |

PS

| F | F | F | 8 | F | 8 | C | 0 |

Memory

<table>
<thead>
<tr>
<th>12345670</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
</tr>
</tbody>
</table>

12345674

Memory

<table>
<thead>
<tr>
<th>12345674</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x x x x x</td>
</tr>
</tbody>
</table>
7.157  STB (Store Byte Data in Register to Memory)

Stores the byte data in Ri to memory address Rj.

- Assembler Format
  
  STB   Ri, @Rj

- Operation
  
  Ri → (Rj)

- Flag Change
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification
  
  Memory Store instruction, Instruction with delay slot

- Execution Cycles
  
  a cycle

- Instruction Format
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**FR81 Family**

- **Execution Example**

```
STB R3, @R2 ; Bit pattern of the instruction: 0001 0110 0010 0011
```

<table>
<thead>
<tr>
<th>R2</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>12345678</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
</table>

Before execution

<table>
<thead>
<tr>
<th>R2</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>12345678</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
</table>

After execution
7.158 STB (Store Byte Data in Register to Memory)

Stores the byte data in Ri to memory address R13 + Rj.

- **Assembler Format**
  
  STB Ri,@(R13, Rj)

- **Operation**
  
  Ri → (R13+Rj)

- **Flag Change**
  
  N, Z, V, C: Unchanged.

- **Classification**
  
  Memory Store instruction, Instruction with delay slot

- **Execution Cycles**
  
  a cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rj</td>
<td></td>
<td>Ri</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**FR81 Family**

- **Execution Example**

  STB R3,@(R13, R2) ; Bit pattern of the instruction: 0001 0010 0010 0011

<table>
<thead>
<tr>
<th></th>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>0 0 0 0 0 0 0 4</td>
<td>0 0 0 0 0 0 0 4</td>
</tr>
<tr>
<td>R3</td>
<td>0 0 0 0 0 0 2 1</td>
<td>0 0 0 0 0 0 2 1</td>
</tr>
<tr>
<td>R13</td>
<td>1 2 3 4 5 6 7 8</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234567B</td>
<td>x x</td>
</tr>
<tr>
<td>1234567C</td>
<td>2 1</td>
</tr>
</tbody>
</table>

Before execution | After execution
7.159 STB (Store Byte Data in Register to Memory)

Stores the byte data in Ri to memory address R14 + o8. The value o8 is a signed calculation. The value of o8 is specified in disp8.

- Assembler Format
  
  STB Ri,@(R14, disp8)

- Operation
  
  Ri → (R14+o8)

- Flag Change
  
  N, Z, V, C: Unchanged.

- Classification
  
  Memory Store instruction, Instruction with delay slot

- Execution Cycles
  
  a cycle

- Instruction Format
  
  MSB | LSB
  -------------------------
  0   1   1   1          o8        Ri
FR81 Family

Execution Example

STB R3, @(R14, 1)  ; Bit pattern of the instruction: 0111 0000 0001 0011

Before execution

<table>
<thead>
<tr>
<th></th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>x x</td>
</tr>
<tr>
<td>12345679</td>
<td>xx</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th></th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>2 1</td>
</tr>
<tr>
<td>12345679</td>
<td>x x</td>
</tr>
</tbody>
</table>
7.160 STB (Store Byte Data in Register to Memory)

Loads the byte data in Ri to memory address BP+u16. Unsigned u16 value is calculated. The value in u16 is specified as udisp16.

- Assembler Format

```assembly
STB   Ri, @(BP, udisp16)
```

- Operation

```
Ri → (BP+u16)
```

- Flag Change

```
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
```

N, Z, V, C: Unchanged.

- Classification

Memory Store instruction, FR81 family

- Execution Cycles

1 cycle

- Instruction Format

```
\begin{array}{cccccccccccc}
\text{MSB} & \text{LSB} \\
| (n+0) | 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 | Ri \\
\hline
| (n+2) | u16 |
\end{array}
```

- EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected.
7.161 STH (Store Halfword Data in Register to Memory)

Stores the half-word data in Ri to memory address Rj.

- **Assembler Format**
  
  STH Ri,@Rj

- **Operation**
  
  Ri → (Rj)

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Memory Store instruction, Instruction with delay slot

- **Execution Cycles**
  
  a cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rj</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ri</td>
</tr>
</tbody>
</table>

  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
Execution Example

```
STH R3, @R2 ; Bit pattern of the instruction: 0001 0101 0010 0011
```

Before execution

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>0 0 0 0 4 3 2 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>x x x x</td>
</tr>
</tbody>
</table>

After execution

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>0 0 0 0 4 3 2 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
</tr>
<tr>
<td>4 3 2 1</td>
</tr>
</tbody>
</table>
7.162 STH (Store Halfword Data in Register to Memory)

Stores the half-word data in Ri to memory address R13 + Rj.

- **Assembler Format**
  
  \[
  \text{STH} \quad \text{Ri,@(R13, Rj)}
  \]

- **Operation**
  
  \[
  \text{Ri} \rightarrow (\text{R13}+\text{Rj})
  \]

- **Flag Change**
  
  \[
  \begin{array}{cccc}
  N & Z & V & C \\
  - & - & - & - \\
  \end{array}
  \]
  
  N, Z, V, C: Unchanged.

- **Classification**
  
  Memory Store instruction, Instruction with delay slot

- **Execution Cycles**
  
  a cycle

- **Instruction Format**
  
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Rj</td>
</tr>
<tr>
<td>Ri</td>
<td></td>
</tr>
</tbody>
</table>
FR81 Family

CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS
7.162

 Execution Example

STH  R3,@(R13, R2)  ; Bit pattern of the instruction: 0001 0001 0010 0011

Before execution

| R2   | 0 0 0 0 0 0 0 4 |
| R3   | 0 0 0 0 4 3 2 1 |
| R13  | 1 2 3 4 5 6 7 8 |
| 1234567A | Memory |
| 1234567C | x x x x |

After execution

| R2   | 0 0 0 0 0 0 0 4 |
| R3   | 0 0 0 0 4 3 2 1 |
| R13  | 1 2 3 4 5 6 7 8 |
| 1234567A | Memory |
| 1234567C | 4 3 2 1 |

Memory

Before execution

1234567A

| x x x x |

After execution

1234567A

| 4 3 2 1 |
7.163 STH (Store Halfword Data in Register to Memory)

Stores the half-word data in Ri to memory address R14 + o8 × 2. The value of o8 × 2 is specified in disp9.

- Assembler Format
  
  STH   Ri,@(R14, disp9)

- Operation

  Ri → (R14+o8×2)

- Flag Change

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification

  Memory Store instruction, Instruction with delay slot

- Execution Cycles

  a cycle

- Instruction Format

  MSB | LSB
  --- | ---
  0   1   0   1   o8   Ri
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

7.163

FR81 Family

Execution Example

STH R3,@(R14,2) ; Bit pattern of the instruction: 0101 0000 0001 0011

R3  0 0 0 0 4 3 2 1
R14 1 2 3 4 5 6 7 8

12345678  Memory  12345678  Memory
1234567A  x x x x  1234567A  4 3 2 1

Before execution  After execution
7.164  STH (Store Halfword Data in Register to Memory)

Loads the half word data in Ri to memory address BP+u16 × 2. Unsigned u16 value is calculated. The value in u16 × 2 is specified as udisp17.

- Assembler Format

  STB Ri, @(BP, udisp17)

- Operation

  Ri → (BP+u16 × 2)

- Flag Change

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification

  Memory Store instruction, FR81 family

- Execution Cycles

  a cycle

- Instruction Format

  MSB
  (n+0) 0 0 0 1 0 1 1 1 0 1 0 1 Ri
  (n+2) u16

- EIT Occurrence and Detection

  A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected.
7.165 STILM (Set Immediate Data to Interrupt Level Mask Register)

Transfers the immediate data to the interrupt level mask register (ILM) in the program status (PS).

- **Assembler Format**
  
  STILM   #u8

- **Operation**
  
  if (ILM < 16)
      u8 → ILM
  else if (u8 < 16)
      u8+16 → ILM
  else
      u8 → ILM

This is a privilege instruction, which is only available in privilege mode. If this instruction is executed in user mode, it causes an invalid instruction exception (privilege instruction execution).

Only the lower 5 bits (bit4 to bit0) of the immediate data are valid. At the time this instruction is executed, if the value of the interrupt level mask register (ILM) is in the range 16 to 31, only new ILM settings between 16 and 31 can be entered. If the value u8 is in the range 0 to 15, the value 16 will be added to that data before being transferred to the ILM. If the original ILM value is in the range 0 to 15, then any value between 0 and 31 can be transferred to the ILM.

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**

  Other instructions, Instruction with delay slot, FR81 updating
FR81 Family

- Execution Cycles
  1 cycle

- Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
<th>u8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- EIT Occurrence and Detection

  User mode:
  Used to generate an invalid instruction exception (privilege instruction execution).

  Privilege mode:
  An interrupt is detected (ILM after instruction execution is used).

- Execution Example

  STILM #14H ; Bit pattern of the instruction: 1000 0111 0001 0100

  ILM       ILM
  ![Before execution](11111) ![After execution](10100)
7.166 STM0 (Store Multiple Registers)

The STM0 instruction stores the word data from multiple registers specified in reglist (from R0 to R7) and repeats the operation of storing the result in address R15 after subtracting the value of 4 from R15. Registers are processed in ascending order.

- **Asmblr Format**
  
  STM0 (reglist)

  Registers from R0 to R7 are separated by ",", arranged and specified in reglist.

- **Operation**

  The following operations are repeated according to the number of registers in reglist.

  
  R15-4 → R15
  
  Ri → (R15)

  The bit values and register numbers for reglist (STM0) are shown in Table 7.166-1.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>R0</td>
</tr>
<tr>
<td>6</td>
<td>R1</td>
</tr>
<tr>
<td>5</td>
<td>R2</td>
</tr>
<tr>
<td>4</td>
<td>R3</td>
</tr>
<tr>
<td>3</td>
<td>R4</td>
</tr>
<tr>
<td>2</td>
<td>R5</td>
</tr>
<tr>
<td>1</td>
<td>R6</td>
</tr>
<tr>
<td>0</td>
<td>R7</td>
</tr>
</tbody>
</table>

- **Flag Change**

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.
FR81 Family

- Classification
  - Other instructions

- Execution Cycles
  - If "n" is the number of registers specified in the parameter reglist, the execution cycles required are as follows.

    When n=0: 1 cycle
    Otherwise: \( a \times n \) cycles

- Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
<th>reglist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Execution Example

  STM0 (R2, R3) ; Bit pattern of the instruction: 1000 1110 0011 0000

  
  
<table>
<thead>
<tr>
<th>R2</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 0 B C 9 3 6 3</td>
<td>9 0 B C 9 3 6 3</td>
</tr>
<tr>
<td>8 3 4 3 8 3 4 A</td>
<td>8 3 4 3 8 3 4 A</td>
</tr>
<tr>
<td>7 F F F F F C 8</td>
<td>7 F F F F F C 0</td>
</tr>
</tbody>
</table>

  
  
  Memory
  
  Before execution
  
  After execution
  
  7FFFFFC0
  x x x x x x x x
  7FFFFFC4
  x x x x x x x x
  7FFFFFC8
  x x x x x x x x
7.167 STM1 (Store Multiple Registers)

The STM1 instruction stores the word data from multiple registers specified in reglist (from R8 to R15) and repeats the operation of storing the result in address R15 after subtracting the value of 4 from R15. Registers are processed in ascending order. If R15 is specified in the parameter reglist, the contents of R15 retained before the instruction is executed will be written to memory.

- Assembler Format
  STM1 (reglist)

  Registers from R0 to R7 are separated by "," , arranged and specified in reglist.

- Operation
  The following operations are repeated according to the number of registers in reglist.

  R15-4 → R15
  Ri → (R15)

  The bit values and register numbers for reglist (STM1) are shown in Table 7.167-1.

Table 7.167-1 Bit values and register numbers for reglist (STM1)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>R8</td>
</tr>
<tr>
<td>6</td>
<td>R9</td>
</tr>
<tr>
<td>5</td>
<td>R10</td>
</tr>
<tr>
<td>4</td>
<td>R11</td>
</tr>
<tr>
<td>3</td>
<td>R12</td>
</tr>
<tr>
<td>2</td>
<td>R13</td>
</tr>
<tr>
<td>1</td>
<td>R14</td>
</tr>
<tr>
<td>0</td>
<td>R15</td>
</tr>
</tbody>
</table>

- Flag Change

  N  Z  V  C
  -------
  -    -  -  -
  N, Z, V, C: Unchanged.
Classify:

- Other instructions

Execution Cycles:

- If "n" is the number of registers specified in the parameter reglist, the execution cycles required are as follows.

  - When \( n=0 \): 1 cycle
  - Otherwise: \( a \times n \) cycles

Instruction Format:

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
<th>reglist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Execution Example:

STM1 (R10, R11, R12) ; Bit pattern of the instruction: 1000 1111 0011 1000

Before execution:

<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FFFFFFC0</td>
<td>x x x x x x x</td>
</tr>
<tr>
<td>7FFFFFFC4</td>
<td>x x x x x x x</td>
</tr>
<tr>
<td>7FFFFFFC8</td>
<td>x x x x x x x</td>
</tr>
<tr>
<td>7FFFFFFCC</td>
<td>x x x x x x x</td>
</tr>
</tbody>
</table>

After execution:

<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FFFFFFC0</td>
<td>8 F E 3 9 E 8 A</td>
</tr>
<tr>
<td>7FFFFFFC4</td>
<td>9 0 B C 9 3 6 3</td>
</tr>
<tr>
<td>7FFFFFFC8</td>
<td>8 D F 7 8 8 E 4</td>
</tr>
<tr>
<td>7FFFFFFCC</td>
<td>x x x x x x x</td>
</tr>
</tbody>
</table>

R10 8 F E 3 9 E 8 A
R11 9 0 B C 9 3 6 3
R12 8 D F 7 8 8 E 4
R15 7 F F F F F C C
7.168 SUB (Subtract Word Data in Source Register from Destination Register)

Subtracts the word data in Rj from the word data in Ri, stores the results to Ri.

- **Assembler Format**
  
  ```
  SUB   Rj, Ri
  ```

- **Operation**
  
  ```
  Ri - Rj → Ri
  ```

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

  - N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  - Z: Set when the operation result is zero, cleared otherwise.
  - V: Set when an overflow has occurred as a result of the operation, cleared otherwise.
  - C: Set when a borrow has occurred as a result of the operation, cleared otherwise.

- **Classification**
  
  Add/Subtract instruction, Instruction with delay slot

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Rj</td>
<td>Ri</td>
</tr>
</tbody>
</table>
### Execution Example

**SUB R2, R3** ; Bit pattern of the instruction: 1010 1100 0010 0011

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
<th>CCR</th>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>9 9 9 9 9 9 9 9</td>
<td>0 0 0 0</td>
<td></td>
<td>1 0 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
<th>CCR</th>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>8 7 6 5 4 3 2 1</td>
<td>1 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.169  **SUBC (Subtract Word Data in Source Register and Carry bit from Destination Register)**

Subtracts word data in Rj and carry flag (C) from Ri, stores the results to Ri.

- **Assembler Format**
  
  SUBC Rj, Ri

- **Operation**
  
  \[ Ri - Rj - C \rightarrow Ri \]

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

  N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".
  Z: Set when the operation result is "0", cleared otherwise.
  V: Set when an overflow has occurred as a result of the operation, cleared otherwise.
  C: Set when a borrow has occurred as a result of the operation, cleared otherwise.

- **Classification**
  
  Add/Subtract instruction, Instruction with delay slot

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Execution Example

```
SUBC R2, R3 ; Bit pattern of the instruction: 1010 1101 0010 0011
```

<table>
<thead>
<tr>
<th>Before execution</th>
<th>After execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>CCR</td>
<td>CCR</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>1 0 0 0</td>
</tr>
</tbody>
</table>
CHAPTER 7  DETAILED EXECUTION INSTRUCTIONS

7.170  SUBN (Subtract Word Data in Source Register from Destination Register)

Subtracts the word data in Rj from the word data in Ri, stores results to Ri without changing the flag settings.

- **Assembler Format**
  
  ```
  SUBN   Rj, Ri
  ```

- **Operation**
  
  ```
  Ri - Rj → Ri
  ```

- **Flag Change**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- **Classification**
  
  Add/Subtract instruction, Instruction with delay slot

- **Execution Cycles**
  
  1 cycle

- **Instruction Format**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>Rj</td>
</tr>
<tr>
<td>Ri</td>
<td></td>
</tr>
</tbody>
</table>
FR81 Family

- Execution Example

```assembly
SUBN R2, R3 ; Bit pattern of the instruction: 1010 1110 0010 0011
```

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
<th>CCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>9 9 9 9 9 9 9 9</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

Before execution

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
<th>CCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>8 7 6 5 4 3 2 1</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

After execution
7.171 XCHB (Exchange Byte Data)

Exchanges the contents of the byte address indicated by Rj and those indicated by Ri. The lower 8 bits of data originally at Ri are transferred to the byte address indicated by Rj and the data originally at Rj is extended with zeros and transferred to Ri.

- Assembler Format
  
  XCHB @Rj, Ri

- Operation
  
  Ri → TEMP
  
  extu((Rj)) → Ri
  
  TEMP → (Rj)

- Flag Change

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  N, Z, V, C: Unchanged.

- Classification

  Other instructions, Read/Modify/Write type instruction

- Execution Cycles

  2a cycles

- Instruction Format

<table>
<thead>
<tr>
<th>MSB</th>
<th></th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Rj</td>
<td></td>
<td>Ri</td>
</tr>
</tbody>
</table>
FR81 Family

Execution Example

XCHB @R1, R0 ; Bit pattern of the instruction: 1000 1010 0001 0000

<table>
<thead>
<tr>
<th>R0</th>
<th>0 0 0 0 0 0 7 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>8 0 0 0 0 0 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>80000001</td>
<td>x x</td>
</tr>
<tr>
<td>80000002</td>
<td>F D</td>
</tr>
<tr>
<td>80000003</td>
<td>x x</td>
</tr>
<tr>
<td>80000001</td>
<td>x x</td>
</tr>
<tr>
<td>80000002</td>
<td>7 8</td>
</tr>
<tr>
<td>80000003</td>
<td>x x</td>
</tr>
</tbody>
</table>

Before execution | After execution
APPENDIX

It includes Instruction Lists and Instruction Maps of FR81 Family.

APPENDIX A  Instruction Lists
APPENDIX B  Instruction Maps
APPENDIX C  Supplemental Explanation about FPU Exception Processing
APPENDIX A Instruction Lists

It includes Instruction Lists of FR81 Family CPU.

A.1 Meaning of Symbols
A.2 Instruction Lists
A.3 List of Instructions that can be positioned in the Delay Slot
FR81 Family

A.1 Meaning of Symbols

This section describes the meaning of symbols used in the Instruction Lists and Detailed Execution Instructions has been explained.

A.1.1 Mnemonic and Operation Columns

These are the symbols used in Mnemonic and Operation columns of Instruction Lists as well as assembler format of Detailed Execution Instructions and operation.

i4

It is 4-bit immediate data. 0(0H) to 15(FH) in case of zero extension and -16(0H) to -1(FH) in case of minus extension can be specified.

Table A.1-1 zero extension and minus extension values of 4-bit immediate data

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Specified Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Zero Extension</td>
</tr>
<tr>
<td>0000B</td>
<td>0</td>
</tr>
<tr>
<td>0001B</td>
<td>1</td>
</tr>
<tr>
<td>0010B</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1101B</td>
<td>13</td>
</tr>
<tr>
<td>1110B</td>
<td>14</td>
</tr>
<tr>
<td>1111B</td>
<td>15</td>
</tr>
</tbody>
</table>

i8

8-bit immediate data, Range 0 (00H) to 255 (FFH)

i20

20-bit immediate data, Range 0 (000000H) to 1,048,575 (FFFFFH)

i32

32-bit immediate data, Range 0 (0000 0000H) to 4,294,967,295 (FFFF FFFFH)

s8

signed 8-bit immediate data, range -128 (80H) to 127 (7FH)

s10

signed 10-bit immediate data, range -512 (200H) to 508 (1FCH) in multiples of 4
APPENDIX
APPENDIX A Instruction Lists

u4
  unsigned 4-bit immediate data, range 0 (0H) to 15 (FH)

u8
  unsigned 8-bit immediate data, range 0 (00H) to 255 (FFH)

u10
  unsigned 10-bit immediate data, range 0 (000H) to 1020 (3FC_H) in multiples of 4

udisp6
  unsigned 6-bit address values, range 0 (00H) to 60 (3CH) in multiples of 4

udisp16
  unsigned 16-bit address values, range 0 (0000H) to 65535 (FFFF_H). range 0 (0000H) to 65532 (FFFC_H) in multiples of 4

udisp17
  unsigned 17-bit address values, range 0 (00000H) to 131070 (1FFFF_H) in multiples of 2

udisp18
  unsigned 18-bit address values, range 0 (00000H) to 262140 (3FFFF_H) in multiples of 4

disp8
  signed 6-bit address values, range -128 (80H) to 127 (7FH)

disp9
  signed 9-bit address values, range -256 (100H) to 254 (0FE_H) in multiples of 2

disp10
  signed 10-bit address values, range -512 (200H) to 508 (1FC_H) in multiples of 4

disp16
  signed 16-bit address values, range -32768 (8000H) to 32764 (FFFCH)

dir8
  Unsigned 8-bit address values, range 0 (0H) to 255 (FFH)
dir9
    unsigned 9-bit address values, range 0 (000H) to 510 (1FEH) in multiples of 2

dir10
    unsigned 10-bit address values, range 0 (000H) to 1020 (3FCH) in multiples of 4

label9
    branch address, range 256 (100H) to 254 (0FEH) in multiples of 2 for the value of Program Counter (PC) +2

label12
    branch address, range -2048 (800H) to 2046 (7FCH) in multiples of 2 for the value of Program Counter (PC) +2

label17
    branch address, range -65536 (10000H) to 65534 (0FFFEH) in multiples of 2 for the value of Program Counter (PC) +2

label21
    branch address, range -1048576 (100000H) to 1048574 (0FFFFEH) in multiples of 2 for the value of Program Counter (PC) +2

rel8
    signed 8-bit relative address. Result which is double the value of rel8 for the value of Program Counter (PC) +2 will denote the Branch Destination Address. Range 128 (80H) to 127 (7FH)

rel11
    signed 11-bit relative address. Result which is double the value of rel11 for the value of Program Counter (PC) +2 will denote the Branch Destination Address. Range -1024 (400H) to 1023 (3FFH)

rel16
    signed 16-bit relative address. Result which is double the value of rel16 for the value of Program Counter (PC) +2 will denote the Branch Destination Address. Range -32768 (8000H) to 32767 (7FFFH)

rel20
    signed 20-bit relative address. Result which is double the value of rel20 for the value of Program Counter (PC) +2 will denote the Branch Destination Address. Range -524288 (80000H) to 524287 (7FFFFFH)
Ri, Rj
Indicates General-purpose Registers (R0 to R15)

### Table A.1-2 Specification of General-purpose register based on Rj/Ri

<table>
<thead>
<tr>
<th>Ri / Rj</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>R0</td>
</tr>
<tr>
<td>0001</td>
<td>R1</td>
</tr>
<tr>
<td>0010</td>
<td>R2</td>
</tr>
<tr>
<td>0011</td>
<td>R3</td>
</tr>
<tr>
<td>0100</td>
<td>R4</td>
</tr>
<tr>
<td>0101</td>
<td>R5</td>
</tr>
<tr>
<td>0110</td>
<td>R6</td>
</tr>
<tr>
<td>0111</td>
<td>R7</td>
</tr>
<tr>
<td>1000</td>
<td>R8</td>
</tr>
<tr>
<td>1001</td>
<td>R9</td>
</tr>
<tr>
<td>1010</td>
<td>R10</td>
</tr>
<tr>
<td>1011</td>
<td>R11</td>
</tr>
<tr>
<td>1100</td>
<td>R12</td>
</tr>
<tr>
<td>1101</td>
<td>R13</td>
</tr>
<tr>
<td>1110</td>
<td>R14</td>
</tr>
<tr>
<td>1111</td>
<td>R15</td>
</tr>
</tbody>
</table>

Rs
Indicates Dedicated Registers (TBR, RP, USP, SSP, MDH, MDL, BP, FCR, ESR, DBR)

### Table A.1-3 Specification of Dedicated Register based on Rs

<table>
<thead>
<tr>
<th>Rs</th>
<th>Register</th>
<th>Rs</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Table Base Register (TBR)</td>
<td>1000</td>
<td>Exception status register (ESR)</td>
</tr>
<tr>
<td>0001</td>
<td>Return Pointer (RP)</td>
<td>1001</td>
<td>Reserved (Disabled)</td>
</tr>
<tr>
<td>0010</td>
<td>System Stack Pointer (SSP)</td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>User Stack Pointer (USP)</td>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>Multiplication/Division Register (MDH)</td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>Multiplication/Division Register (MDL)</td>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>Base pointer (BP)</td>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>FPU control register (FCR)</td>
<td>1111</td>
<td>Debug register (DBR)</td>
</tr>
</tbody>
</table>


FR81 Family

FRi, FRj, FRk
Indicates Floating Point Registers (FR0 to FR15)

Table A.1-4 Floating Point Register based on FRi/FRj/FRk

<table>
<thead>
<tr>
<th>FRi/FRj/FRk</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>FR0</td>
</tr>
<tr>
<td>0001</td>
<td>FR1</td>
</tr>
<tr>
<td>0010</td>
<td>FR2</td>
</tr>
<tr>
<td>0011</td>
<td>FR3</td>
</tr>
<tr>
<td>0100</td>
<td>FR4</td>
</tr>
<tr>
<td>0101</td>
<td>FR5</td>
</tr>
<tr>
<td>0110</td>
<td>FR6</td>
</tr>
<tr>
<td>0111</td>
<td>FR7</td>
</tr>
</tbody>
</table>

FRi/FRj/FRk Register

<table>
<thead>
<tr>
<th>FRi/FRj/FRk</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>FR8</td>
</tr>
<tr>
<td>1001</td>
<td>FR9</td>
</tr>
<tr>
<td>1010</td>
<td>FR10</td>
</tr>
<tr>
<td>1011</td>
<td>FR11</td>
</tr>
<tr>
<td>1100</td>
<td>FR12</td>
</tr>
<tr>
<td>1101</td>
<td>FR13</td>
</tr>
<tr>
<td>1110</td>
<td>FR14</td>
</tr>
<tr>
<td>1111</td>
<td>FR15</td>
</tr>
</tbody>
</table>

(reglist)
Indicates 8-bit Register list. General purpose register corresponding to each bit value can be specified.

Table A.1-5 Correspondence between reglist of LDM0,LDM1 Instruction and General purpose Register

<table>
<thead>
<tr>
<th>LDM0 Instruction</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit0</td>
<td>R0</td>
</tr>
<tr>
<td>bit1</td>
<td>R1</td>
</tr>
<tr>
<td>bit2</td>
<td>R2</td>
</tr>
<tr>
<td>bit3</td>
<td>R3</td>
</tr>
<tr>
<td>bit4</td>
<td>R4</td>
</tr>
<tr>
<td>bit5</td>
<td>R5</td>
</tr>
<tr>
<td>bit6</td>
<td>R6</td>
</tr>
<tr>
<td>bit7</td>
<td>R7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LDM1 Instruction</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit0</td>
<td>R8</td>
</tr>
<tr>
<td>bit1</td>
<td>R9</td>
</tr>
<tr>
<td>bit2</td>
<td>R10</td>
</tr>
<tr>
<td>bit3</td>
<td>R11</td>
</tr>
<tr>
<td>bit4</td>
<td>R12</td>
</tr>
<tr>
<td>bit5</td>
<td>R13</td>
</tr>
<tr>
<td>bit6</td>
<td>R14</td>
</tr>
<tr>
<td>bit7</td>
<td>R15</td>
</tr>
</tbody>
</table>

Table A.1-6 Correspondence between reglist of STM0,STM1 Instruction and General purpose Register

<table>
<thead>
<tr>
<th>STM0 Instruction</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit0</td>
<td>R7</td>
</tr>
<tr>
<td>bit1</td>
<td>R6</td>
</tr>
<tr>
<td>bit2</td>
<td>R5</td>
</tr>
<tr>
<td>bit3</td>
<td>R4</td>
</tr>
<tr>
<td>bit4</td>
<td>R3</td>
</tr>
<tr>
<td>bit5</td>
<td>R2</td>
</tr>
<tr>
<td>bit6</td>
<td>R1</td>
</tr>
<tr>
<td>bit7</td>
<td>R0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STM1 Instruction</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit0</td>
<td>R15</td>
</tr>
<tr>
<td>bit1</td>
<td>R14</td>
</tr>
<tr>
<td>bit2</td>
<td>R13</td>
</tr>
<tr>
<td>bit3</td>
<td>R12</td>
</tr>
<tr>
<td>bit4</td>
<td>R11</td>
</tr>
<tr>
<td>bit5</td>
<td>R10</td>
</tr>
<tr>
<td>bit6</td>
<td>R9</td>
</tr>
<tr>
<td>bit7</td>
<td>R8</td>
</tr>
</tbody>
</table>
APPENDIX
APPENDIX A  Instruction Lists

**FR81 Family**

(frlst)

Indicates 16-bit Register list. Floating Point Registers (FR0 to FR15) corresponding to each bit value can be specified.

**Table A.1-7 Correspondence between frlist bit of FLDM Instruction and Floating Point Registers**

<table>
<thead>
<tr>
<th>frlist</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit0</td>
<td>FR0</td>
</tr>
<tr>
<td>bit1</td>
<td>FR1</td>
</tr>
<tr>
<td>bit2</td>
<td>FR2</td>
</tr>
<tr>
<td>bit3</td>
<td>FR3</td>
</tr>
<tr>
<td>bit4</td>
<td>FR4</td>
</tr>
<tr>
<td>bit5</td>
<td>FR5</td>
</tr>
<tr>
<td>bit6</td>
<td>FR6</td>
</tr>
<tr>
<td>bit7</td>
<td>FR7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>frlist</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit8</td>
<td>FR8</td>
</tr>
<tr>
<td>bit9</td>
<td>FR9</td>
</tr>
<tr>
<td>bit10</td>
<td>FR10</td>
</tr>
<tr>
<td>bit11</td>
<td>FR11</td>
</tr>
<tr>
<td>bit12</td>
<td>FR12</td>
</tr>
<tr>
<td>bit13</td>
<td>FR13</td>
</tr>
<tr>
<td>bit14</td>
<td>FR14</td>
</tr>
<tr>
<td>bit15</td>
<td>FR15</td>
</tr>
</tbody>
</table>

**Table A.1-8 Correspondence between frlist bit of FSTM Instruction and Floating Point Registers**

<table>
<thead>
<tr>
<th>frlist</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit0</td>
<td>FR15</td>
</tr>
<tr>
<td>bit1</td>
<td>FR14</td>
</tr>
<tr>
<td>bit2</td>
<td>FR13</td>
</tr>
<tr>
<td>bit3</td>
<td>FR12</td>
</tr>
<tr>
<td>bit4</td>
<td>FR11</td>
</tr>
<tr>
<td>bit5</td>
<td>FR10</td>
</tr>
<tr>
<td>bit6</td>
<td>FR9</td>
</tr>
<tr>
<td>bit7</td>
<td>FR8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>frlist</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit8</td>
<td>FR7</td>
</tr>
<tr>
<td>bit9</td>
<td>FR6</td>
</tr>
<tr>
<td>bit10</td>
<td>FR5</td>
</tr>
<tr>
<td>bit11</td>
<td>FR4</td>
</tr>
<tr>
<td>bit12</td>
<td>FR3</td>
</tr>
<tr>
<td>bit13</td>
<td>FR2</td>
</tr>
<tr>
<td>bit14</td>
<td>FR1</td>
</tr>
<tr>
<td>bit15</td>
<td>FR0</td>
</tr>
</tbody>
</table>

**A.1.2 Operation Column**

These are symbols used in Operation Column of Instruction Lists and operation of Detailed Execution Instructions.

extu( )

indicates a zero extension operation, in which portion lacking higher bits is complimented by adding "0" bit.

extn( )

indicates a minus extension operation, in which portion lacking higher bits is complimented by adding "1" bit.

exts( )

indicates a sign extension operation, in which zero extension is performed for the data within ( ) if MSB
is "0" and a minus extension is performed if MSB is "1".

&
Indicates logical calculation of each bit (AND)

|
Indicates the logical sum of each bit (OR)

^  
Indicates Dedicated Logical Sum of each bit (EXOR)

()  
Indicates specification of indirect address. It is address memory read/write value of the Register or formula within ( ).

{ }  
Indicate the calculation priority. Since ( ) is used for specifying indirect address, different bracket namely { } is used.

if (Condition) then {formula} or if (condition) then {Formula 1} else {Formula 2}
Indicates the execution of conditions. If the conditions are established, formula after ‘then’ is executed and when the conditions are not established, formula next to ‘else’ is executed. Formula can be described variously using the { }.

[m:n]
Bits from m to n are extracted and targeted for operation.

### A.1.3 Format Column
Symbols used in the Format Column of the Instruction Lists.

A to N
Indicates the Instruction Formats. A to N correspond to TYPE-A to TYPE-N.

### A.1.4 OP Column
Hexadecimal value used in the Instruction Lists. They denote operation codes (OP). They branch into the following depending on the Instruction Format.

TYPE-A, TYPE-C, TYPE-D, TYPE-G
2-digit hexadecimal value represents 8-bit OP code
APPENDIX A Instruction Lists

TYPE-B
2-digit hexadecimal value represents 4 bits of OP code as higher 1 digit and "0" for lower digit.

TYPE-E, TYPE-H, TYPE-I, TYPE-J, TYPE-K
3-digit hexadecimal value represents 12-bit OP code.

TYPE-F
2-digit hexadecimal value represents 8 bits with 3-bit 000B added below 5-bit OP code.

TYPE-L, TYPE-N
4-digit hexadecimal value represents 16 bits with 2-bit 00B added below 14-bit OP code.

TYPE-M
4-digit hexadecimal value represents 16-bit OP code.

A.1.5 CYC Column
Symbols used in CYC Column of Instruction Lists and execution cycles of Detailed Execution Instructions. Numerical values represent CPU clock cycles. Minimum of a to d is 1 cycle.

a
Memory access cycles. Cycles change depending on the access target. Minimum value is 1 cycle.

b
Memory access cycles. Cycles change depending on the access target. Minimum value is 1 cycle.

It is 1 cycle when uncompleted LD Instructions are less than 4 Instructions and Register which is the object of load operation is not referred by the subsequent Instruction.

When uncompleted LD Instructions become more than 4 in number, an interlock will be applied from that point till the completion of first LD Instruction and the number of execution cycles will be increased by (Memory Access Cycles - Number of cycles from the issue of an Instruction till first LD Instruction is completed).

When the Register which is target of load operation is referred to by the succeeding Instruction, an interlock will be applied from that point and the number of execution cycles will increase by (Memory Access Cycles - Number of cycle from the issue of an Instruction till an instruction refers to the targeted Register + 1).
c
An interlock will be applied when the immediately next Instruction refers to Multiplication/Division
Register (MDH) and the number of execution cycles will be increased to 2. Otherwise it will be 1
cycle.

d
There will be 2 cycles when pre-fetching of Instruction in the Pre-fetch Buffer is not carried out.
Minimum value is 1 cycle.

A.1.6 FLAG Column
Symbols used for flag change in the Flag Column of Instruction Lists and Detailed Execution Instructions.
Represents change in Negative Flag (N), Zero Flag (Z), Overflow Flag (V), Carry Flag (C) of the Condition
Code Register (CCR).

C
Varies depending on the result of operation

-  No change

0  Value becomes "0"

1  Value becomes "1"

A.1.7 RMW Column
Symbols used in the RMW Column of Instruction Lists. It represents whether or not it is Read-Modify-
Write Instruction.

-   Instruction is not Read-Modify-Write Instruction.

○  Instruction is Read-Modify-Write Instruction.

A.1.8 Reference Column
Represents the portion explained in “Chapter 7 Detailed Execution Instructions”
This part indicates Instruction Lists of FR81 Family CPU.

There are a total of 231 instructions in FR81 Family CPU. These instructions are divided into the following 21 categories.

- Add/Subtract Instructions (10 Instructions)
- Compare Calculation Instructions (3 Instructions)
- Logical Calculation Instructions (12 Instructions)
- Bit Operation Instructions (8 Instructions)
- Multiply/Divide Instructions (10 Instructions)
- Shift Instructions (9 Instructions)
- Immediate Data Transfer Instructions (3 Instructions)
- Memory Load Instructions (16 Instructions)
- Memory Store Instructions (16 Instructions)
- Inter-Register Transfer Instructions/Dedicated Register Transfer Instructions (5 Instructions)
- Non-delayed Branching Instructions (24 Instructions)
- Delayed Branching Instructions (21 Instructions)
- Direct Addressing Instructions (14 Instructions)
- Bit Search Instructions (3 Instructions)
- Other Instructions (16 Instructions)
- FPU Memory Load Instructions (7 Instructions)
- FPU Memory Store Instructions (7 Instructions)
- FPU Single-Precision Floating Point Calculation Instruction (12 Instructions)
- FPU Inter-Register Transfer Instruction (3 Instructions)
- FPU Branching Instruction without Delay (16 Instructions)
- FPU Branching Instruction with Delay (16 Instructions)
## Table A.2-1  Add/Subtract Instructions (10 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FLAG</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>C</td>
<td>A6</td>
<td>1</td>
<td>CCCC</td>
<td>-</td>
<td>Ri+Rj → Ri</td>
<td></td>
<td>7.2</td>
</tr>
<tr>
<td>ADD #i4, Ri</td>
<td>C</td>
<td>A4</td>
<td>1</td>
<td>CCCC</td>
<td>-</td>
<td>Ri+extu(i4) → Ri</td>
<td>i4 is zero extension</td>
<td>7.1</td>
</tr>
<tr>
<td>ADD2 #i4, Ri</td>
<td>C</td>
<td>A5</td>
<td>1</td>
<td>CCCC</td>
<td>-</td>
<td>Ri+extn(i4) → Ri</td>
<td>i4 is Minus extension</td>
<td>7.3</td>
</tr>
<tr>
<td>ADC</td>
<td>A</td>
<td>A7</td>
<td>1</td>
<td>CCCC</td>
<td>-</td>
<td>Ri+Rj+C → Ri</td>
<td>Add with carry</td>
<td>7.4</td>
</tr>
<tr>
<td>ADDN</td>
<td>A</td>
<td>A2</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>Ri+Rj → Ri</td>
<td></td>
<td>7.6</td>
</tr>
<tr>
<td>ADDN #i4, Ri</td>
<td>C</td>
<td>A0</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>Ri+extu(i4) → Ri</td>
<td>i4 is Zero extension</td>
<td>7.5</td>
</tr>
<tr>
<td>ADDN2 #i4, Ri</td>
<td>C</td>
<td>A1</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>Ri+extn(i4) → Ri</td>
<td>i4 is Minus extension</td>
<td>7.7</td>
</tr>
<tr>
<td>SUB</td>
<td>A</td>
<td>AC</td>
<td>1</td>
<td>CCCC</td>
<td>-</td>
<td>Ri-Rj → Ri</td>
<td></td>
<td>7.129</td>
</tr>
<tr>
<td>SUBC</td>
<td>A</td>
<td>AD</td>
<td>1</td>
<td>CCCC</td>
<td>-</td>
<td>Ri-Rj-C → Ri</td>
<td>Add with carry</td>
<td>7.130</td>
</tr>
<tr>
<td>SUBN</td>
<td>A</td>
<td>AE</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>Ri-Rj → Ri</td>
<td></td>
<td>7.131</td>
</tr>
</tbody>
</table>

## Table A.2-2  Compare Calculation Instructions (3 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FLAG</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td>C</td>
<td>A8</td>
<td>1</td>
<td>CCCC</td>
<td>-</td>
<td>Ri-Rj</td>
<td></td>
<td>7.32</td>
</tr>
<tr>
<td>CMP #i4, Ri</td>
<td>C</td>
<td>A9</td>
<td>1</td>
<td>CCCC</td>
<td>-</td>
<td>Ri-extu(i4)</td>
<td>i4 is Zero extension</td>
<td>7.31</td>
</tr>
<tr>
<td>CMP2 #i4, Ri</td>
<td>C</td>
<td>A1</td>
<td>1</td>
<td>CCCC</td>
<td>-</td>
<td>Ri-extn(i4)</td>
<td>i4 is Minus extension</td>
<td>7.33</td>
</tr>
</tbody>
</table>

## Table A.2-3  Logical Calculation Instructions (12 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FLAG</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>A</td>
<td>82</td>
<td>1</td>
<td>CC--</td>
<td>-</td>
<td>Ri &amp; Rj → Ri</td>
<td>Word</td>
<td>7.10</td>
</tr>
<tr>
<td>AND</td>
<td>A</td>
<td>84</td>
<td>1+2a</td>
<td>CC--</td>
<td>0</td>
<td>(Ri) &amp; (Ri)</td>
<td>Word</td>
<td>7.9</td>
</tr>
<tr>
<td>ANDH</td>
<td>A</td>
<td>85</td>
<td>1+2a</td>
<td>CC--</td>
<td>0</td>
<td>(Ri) &amp; (Ri)</td>
<td>Half-Word</td>
<td>7.13</td>
</tr>
<tr>
<td>ANDB</td>
<td>A</td>
<td>86</td>
<td>1+2a</td>
<td>CC--</td>
<td>0</td>
<td>(Ri) &amp; (Ri)</td>
<td>Byte</td>
<td>7.11</td>
</tr>
<tr>
<td>OR</td>
<td>A</td>
<td>92</td>
<td>1</td>
<td>CC--</td>
<td>-</td>
<td>Ri</td>
<td>Rj → Rj</td>
<td>Word</td>
</tr>
<tr>
<td>OR</td>
<td>A</td>
<td>94</td>
<td>1+2a</td>
<td>CC--</td>
<td>0</td>
<td>(Ri)</td>
<td>Rj → Rj</td>
<td>Word</td>
</tr>
<tr>
<td>ORH</td>
<td>A</td>
<td>95</td>
<td>1+2a</td>
<td>CC--</td>
<td>0</td>
<td>(Ri)</td>
<td>Rj → Rj</td>
<td>Half-Word</td>
</tr>
<tr>
<td>ORB</td>
<td>A</td>
<td>96</td>
<td>1+2a</td>
<td>CC--</td>
<td>0</td>
<td>(Ri)</td>
<td>Rj → Rj</td>
<td>Byte</td>
</tr>
<tr>
<td>EOR</td>
<td>A</td>
<td>9A</td>
<td>1</td>
<td>CC--</td>
<td>-</td>
<td>Ri ^ Rj → Rj</td>
<td>Word</td>
<td>7.56</td>
</tr>
<tr>
<td>EOR</td>
<td>A</td>
<td>9C</td>
<td>1+2a</td>
<td>CC--</td>
<td>0</td>
<td>(Ri) ^ Rj → Rj</td>
<td>Word</td>
<td>7.55</td>
</tr>
<tr>
<td>EORH</td>
<td>A</td>
<td>9D</td>
<td>1+2a</td>
<td>CC--</td>
<td>0</td>
<td>(Ri) ^ Rj → Rj</td>
<td>Half-Word</td>
<td>7.58</td>
</tr>
<tr>
<td>EORB</td>
<td>A</td>
<td>9E</td>
<td>1+2a</td>
<td>CC--</td>
<td>0</td>
<td>(Ri) ^ Rj → Rj</td>
<td>Byte</td>
<td>7.57</td>
</tr>
</tbody>
</table>
### APPENDIX A Instruction Lists

#### Table A.2-4 Bit Operation Instructions (8 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FLAG NZVC</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANDL #u4, @Ri</td>
<td>C</td>
<td>80</td>
<td>1+2a</td>
<td></td>
<td></td>
<td>(Ri) &amp; {F0H+u4} → (Ri)</td>
<td>Lower 4-bit</td>
<td>7.18</td>
</tr>
<tr>
<td>BANDH #u4, @Ri</td>
<td>C</td>
<td>81</td>
<td>1+2a</td>
<td></td>
<td></td>
<td>(Ri) &amp; [u4&lt;u4+0FH] → (Ri)</td>
<td>Higher 4-bit</td>
<td>7.17</td>
</tr>
<tr>
<td>BORL #u4, @Ri</td>
<td>C</td>
<td>90</td>
<td>1+2a</td>
<td></td>
<td></td>
<td>(Ri)</td>
<td>u4 → (Ri)</td>
<td>Lower 4-bit</td>
</tr>
<tr>
<td>BORH #u4, @Ri</td>
<td>C</td>
<td>91</td>
<td>1+2a</td>
<td></td>
<td></td>
<td>(Ri)</td>
<td>{u4&lt;&lt;4} → (Ri)</td>
<td>Higher 4-bit</td>
</tr>
<tr>
<td>BEORL #u4, @Ri</td>
<td>C</td>
<td>98</td>
<td>1+2a</td>
<td></td>
<td></td>
<td>(Ri) ^ u4 → (Ri)</td>
<td>Lower 4-bit</td>
<td>7.22</td>
</tr>
<tr>
<td>BEORH #u4, @Ri</td>
<td>C</td>
<td>99</td>
<td>1+2a</td>
<td></td>
<td></td>
<td>(Ri) ^ {u4&lt;&lt;4} → (Ri)</td>
<td>Higher 4-bit</td>
<td>7.21</td>
</tr>
<tr>
<td>BTSTL #u4, @Ri</td>
<td>C</td>
<td>88</td>
<td>2+a</td>
<td>0C--</td>
<td></td>
<td>(Ri) &amp; u4</td>
<td>Lower 4-bit</td>
<td>7.26</td>
</tr>
<tr>
<td>BTSTH #u4, @Ri</td>
<td>C</td>
<td>89</td>
<td>2+a</td>
<td>CC--</td>
<td></td>
<td>(Ri) &amp; {u4&lt;&lt;4}</td>
<td>Higher 4-bit</td>
<td>7.25</td>
</tr>
</tbody>
</table>

#### Table A.2-5 Multiply/Divide Instructions (10 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FLAG NZVC</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL Rj, Ri</td>
<td>A</td>
<td>AF</td>
<td>5</td>
<td>CCC--</td>
<td></td>
<td>Ri × Rj → MDH,MDL</td>
<td>32 × 32 bit = 64 bit</td>
<td>7.133</td>
</tr>
<tr>
<td>MULU Rj, Ri</td>
<td>A</td>
<td>AB</td>
<td>5</td>
<td>CCC--</td>
<td></td>
<td>Ri × Rj → MDH,MDL</td>
<td>Unsigned</td>
<td>7.135</td>
</tr>
<tr>
<td>MULH Rj, Ri</td>
<td>A</td>
<td>BF</td>
<td>3</td>
<td>CC--</td>
<td></td>
<td>Ri × Rj → MDL</td>
<td>16 × 16 bit = 32 bit</td>
<td>7.134</td>
</tr>
<tr>
<td>MULUH Rj, Ri</td>
<td>A</td>
<td>BB</td>
<td>3</td>
<td>CC--</td>
<td></td>
<td>Ri × Rj → MDL</td>
<td>Unsigned</td>
<td>7.136</td>
</tr>
<tr>
<td>DIV0S Ri</td>
<td>E</td>
<td>97-4</td>
<td>1</td>
<td>----</td>
<td></td>
<td>MDL ÷ Ri → MDL</td>
<td>In the Specified Instruction Sequence</td>
<td>7.34</td>
</tr>
<tr>
<td>DIV0U Ri</td>
<td>E</td>
<td>97-5</td>
<td>1</td>
<td>----</td>
<td></td>
<td>MDL%Ri → MDH</td>
<td>Step Calculation</td>
<td>7.36</td>
</tr>
<tr>
<td>DIV1 Ri</td>
<td>E</td>
<td>97-6</td>
<td>1</td>
<td>-C-C</td>
<td></td>
<td>MDL + Ri → MDL</td>
<td>32 ÷ 32 bit = 32 bit</td>
<td>7.37</td>
</tr>
<tr>
<td>DIV2 Ri</td>
<td>E</td>
<td>97-7</td>
<td>c</td>
<td>-C-C</td>
<td></td>
<td>MDL%Ri → MDH</td>
<td></td>
<td>7.38</td>
</tr>
<tr>
<td>DIV3</td>
<td>E'</td>
<td>9F-6</td>
<td>1</td>
<td>----</td>
<td></td>
<td></td>
<td></td>
<td>7.39</td>
</tr>
<tr>
<td>DIV4S</td>
<td>E'</td>
<td>9F-7</td>
<td>1</td>
<td>----</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Table A.2-6 Shift Instructions (9 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FLAG NZVC</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL Rj, Ri</td>
<td>A</td>
<td>B6</td>
<td>1</td>
<td>CC-C</td>
<td></td>
<td>Ri &lt;&lt; Rj → Ri</td>
<td>Logical Shift</td>
<td>7.120</td>
</tr>
<tr>
<td>LSL #u4, Ri</td>
<td>C</td>
<td>B4</td>
<td>1</td>
<td>CC-C</td>
<td></td>
<td>Ri &lt;&lt; u4 → Ri</td>
<td>Logical Shift</td>
<td>7.121</td>
</tr>
<tr>
<td>LSL2 #u4, Ri</td>
<td>C</td>
<td>B5</td>
<td>1</td>
<td>CC-C</td>
<td></td>
<td>Ri &lt;&lt; {u4+16} → Ri</td>
<td>Logical Shift</td>
<td>7.122</td>
</tr>
<tr>
<td>LSR Rj, Ri</td>
<td>A</td>
<td>B2</td>
<td>1</td>
<td>CC-C</td>
<td></td>
<td>Ri &gt;&gt; Rj → Ri</td>
<td>Logical Shift</td>
<td>7.123</td>
</tr>
<tr>
<td>LSR #u4, Ri</td>
<td>C</td>
<td>B0</td>
<td>1</td>
<td>CC-C</td>
<td></td>
<td>Ri &gt;&gt; u4 → Ri</td>
<td>Logical Shift</td>
<td>7.124</td>
</tr>
<tr>
<td>LSR2 #u4, Ri</td>
<td>C</td>
<td>B1</td>
<td>1</td>
<td>CC-C</td>
<td></td>
<td>Ri &gt;&gt; {u4+16} → Ri</td>
<td>Arithmetic Shift</td>
<td>7.125</td>
</tr>
<tr>
<td>ASR Rj, Ri</td>
<td>A</td>
<td>BA</td>
<td>1</td>
<td>CC-C</td>
<td></td>
<td>Ri &gt;&gt; Rj → Ri</td>
<td>Arithmetic Shift</td>
<td>7.14</td>
</tr>
<tr>
<td>ASR #u4, Ri</td>
<td>C</td>
<td>B8</td>
<td>1</td>
<td>CC-C</td>
<td></td>
<td>Ri &gt;&gt; u4 → Ri</td>
<td>Arithmetic Shift</td>
<td>7.15</td>
</tr>
<tr>
<td>ASR2 #u4, Ri</td>
<td>C</td>
<td>B9</td>
<td>1</td>
<td>CC-C</td>
<td></td>
<td>Ri &gt;&gt; {u4+16} → Ri</td>
<td>Arithmetic Shift</td>
<td>7.16</td>
</tr>
</tbody>
</table>

#### Table A.2-7 Immediate Data Transfer Instructions (3 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FLAG NZVC</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDI:32 #i32, Ri</td>
<td>H</td>
<td>9F-8</td>
<td>d</td>
<td>----</td>
<td></td>
<td>i32 → Ri</td>
<td></td>
<td>7.107</td>
</tr>
<tr>
<td>LDI:20 #i20, Ri</td>
<td>G</td>
<td>9B</td>
<td>d</td>
<td>----</td>
<td></td>
<td>extu(i20) → Ri</td>
<td>Higher 12-Bits are Zero extension</td>
<td>7.106</td>
</tr>
<tr>
<td>LDI:8 #i8, Ri</td>
<td>B</td>
<td>C0</td>
<td>1</td>
<td>----</td>
<td></td>
<td>extu(i8) → Ri</td>
<td>Higher 24-Bits are Zero extension</td>
<td>7.108</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Format</td>
<td>OP</td>
<td>CYC</td>
<td>FLAG</td>
<td>NZVC</td>
<td>RMW</td>
<td>Operation</td>
<td>Remarks</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>----</td>
<td>-----</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>LD @Rj, Ri</td>
<td>A</td>
<td>04</td>
<td>b</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>(Rj) → Ri</td>
<td></td>
</tr>
<tr>
<td>LD @(R13, Rj), Ri</td>
<td>A</td>
<td>00</td>
<td>b</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>(R13+Rj) → Ri</td>
<td></td>
</tr>
<tr>
<td>LD @(R14, disp10), Ri</td>
<td>B</td>
<td>20</td>
<td>b</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>(R14+o8 × 4) → Ri</td>
<td></td>
</tr>
<tr>
<td>LD @(R15, udisp6), Ri</td>
<td>C</td>
<td>03</td>
<td>b</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>(R15+u4 × 4) → Ri</td>
<td></td>
</tr>
<tr>
<td>LD @R15+, Ri</td>
<td>E</td>
<td>07-0</td>
<td>b</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>(R15) → Ri, R15+4 → R15</td>
<td></td>
</tr>
<tr>
<td>LD @R15+, Rs</td>
<td>E</td>
<td>07-8</td>
<td>b</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>(R15) → Rs, R15+4 → R15</td>
<td></td>
</tr>
<tr>
<td>LD @R15+, PS</td>
<td>E</td>
<td>07-9</td>
<td>l+a</td>
<td>CCCC</td>
<td>-</td>
<td>(R15) → PS, R15+4 → R15</td>
<td></td>
<td>7.105</td>
</tr>
<tr>
<td>LD @(BP, udisp18), Ri</td>
<td>J</td>
<td>07-4</td>
<td>a</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>(BP+u16 × 4) → Ri</td>
<td></td>
</tr>
<tr>
<td>LDUH @Rj, Ri</td>
<td>A</td>
<td>05</td>
<td>b</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>extu((Rj)) → Ri</td>
<td></td>
</tr>
<tr>
<td>LDUH @(R13, Rj), Ri</td>
<td>A</td>
<td>01</td>
<td>b</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>extu((R13+Rj)) → Ri</td>
<td></td>
</tr>
<tr>
<td>LDUH @(R14, disp9), Ri</td>
<td>B</td>
<td>40</td>
<td>b</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>extu((R14+o8 × 2)) → Rj</td>
<td></td>
</tr>
<tr>
<td>LDUH @(BP, udisp17), Ri</td>
<td>J</td>
<td>07-5</td>
<td>a</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>(BP+u16 × 2) → Ri</td>
<td></td>
</tr>
<tr>
<td>LDUB @Rj, Ri</td>
<td>A</td>
<td>06</td>
<td>b</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>extu((Rj)) → Ri</td>
<td></td>
</tr>
<tr>
<td>LDUB @(R13, Rj), Ri</td>
<td>A</td>
<td>02</td>
<td>b</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>extu((R13+Rj)) → Ri</td>
<td></td>
</tr>
<tr>
<td>LDUB @(R14, disp8), Ri</td>
<td>B</td>
<td>60</td>
<td>b</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>extu((R14+o8)) → Ri</td>
<td></td>
</tr>
<tr>
<td>LDUB @(BP, udisp16), Ri</td>
<td>J</td>
<td>07-6</td>
<td>a</td>
<td>----</td>
<td>----</td>
<td>-</td>
<td>(BP+u16) → Ri</td>
<td></td>
</tr>
</tbody>
</table>

- Relation of field o8 in the Instruction Format TYPE-B to the values disp8 to disp10 in assembly notation is as follows.
  
  o8 = disp8
  o8 = disp9 >> 1
  o8 = disp10 >> 2

- Relation of field u4 in the Instruction Format TYPE-C to the values udisp6 in assembly notation is as follows.
  
  u4 = udisp6 >> 2

- Relation of field u16 in the Instruction Format TYPE-J to the values udisp16 to udisp18 in assembly notation is as follows.
  
  u16 = udisp16
  u16 = udisp17 >> 1
  u16 = udisp18 >> 2
### Table A.2-9 Memory Store Instructions (16 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FLAG</th>
<th>NZVC</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST Ri, @Rj</td>
<td>A</td>
<td>14</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>Ri → (Rj)</td>
<td></td>
<td>7.149</td>
</tr>
<tr>
<td>ST Ri, @(R13, Rj)</td>
<td>A</td>
<td>10</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>Ri → (R13+Rj)</td>
<td></td>
<td>7.150</td>
</tr>
<tr>
<td>ST Ri, @(R14, disp10)</td>
<td>B</td>
<td>30</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>Ri → (R14+o8 × 4)</td>
<td></td>
<td>7.151</td>
</tr>
<tr>
<td>ST Ri, @(R15, udisp6)</td>
<td>C</td>
<td>13</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>Ri → (R15+u4 × 4)</td>
<td></td>
<td>7.152</td>
</tr>
<tr>
<td>ST Ri, @-R15</td>
<td>E</td>
<td>17-0</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>R15-4 → R15, Ri → (R15)</td>
<td></td>
<td>7.153</td>
</tr>
<tr>
<td>ST Rs, @-R15</td>
<td>E</td>
<td>17-8</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>R15-4 → R15, Rs → (R15)</td>
<td></td>
<td>7.155</td>
</tr>
<tr>
<td>ST PS, @-R15</td>
<td>E</td>
<td>17-9</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>R15-4 → R15, PS → (R15)</td>
<td></td>
<td>7.156</td>
</tr>
<tr>
<td>ST Ri, @(BP, udisp18)</td>
<td>J</td>
<td>17-4</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>Ri → (BP+u16 × 4)</td>
<td></td>
<td>7.154</td>
</tr>
<tr>
<td>STH Ri, @Rj</td>
<td>A</td>
<td>15</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>Ri → (Rj)</td>
<td></td>
<td>7.161</td>
</tr>
<tr>
<td>STH Ri, @(R13, Rj)</td>
<td>A</td>
<td>11</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>Ri → (R13+Rj)</td>
<td></td>
<td>7.162</td>
</tr>
<tr>
<td>STH Ri, @(R14, disp9)</td>
<td>B</td>
<td>50</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>Ri → (R14+o8 × 2)</td>
<td></td>
<td>7.163</td>
</tr>
<tr>
<td>STH Ri, @(BP, udisp17)</td>
<td>J</td>
<td>17-5</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>Ri → (BP+u16 × 2)</td>
<td></td>
<td>7.164</td>
</tr>
<tr>
<td>STB Ri, @Rj</td>
<td>A</td>
<td>16</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>Ri → (Rj)</td>
<td></td>
<td>7.157</td>
</tr>
<tr>
<td>STB Ri, @(R13, Rj)</td>
<td>A</td>
<td>12</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>Ri → (R13+Rj)</td>
<td></td>
<td>7.158</td>
</tr>
<tr>
<td>STB Ri, @(R14, disp8)</td>
<td>B</td>
<td>70</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>Ri → (R14+o8)</td>
<td></td>
<td>7.159</td>
</tr>
<tr>
<td>STB Ri, @(BP, udisp16)</td>
<td>J</td>
<td>17-6</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td></td>
<td>Ri → (BP+u16)</td>
<td></td>
<td>7.160</td>
</tr>
</tbody>
</table>

- Relation of field o8 in the Instruction Format TYPE-B to the values disp8 to disp10 in assembly notation is as follows.
  - o8 = disp8
  - o8 = disp9 >> 1
  - o8 = disp10 >> 2

- Relation of field u4 in the Instruction Format TYPE-C to the values udisp6 in assembly notation is as follows.
  - u4 = udisp6 >> 2

- Relation of field u16 in the Instruction Format TYPE-J to the values udisp16 to udisp18 in assembly notation is as follows.
  - u16 = udisp16
  - u16 = udisp17 >> 1
  - u16 = udisp18 >> 2
### FR81 Family

**Table A.2-10 Inter-Register Transfer Instructions/Dedicated Register Transfer Instructions (5 Instructions)**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FLAG</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV Rj, Ri</td>
<td>A</td>
<td>8B</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>Rj → Ri</td>
<td>Transfer between general-purpose Registers</td>
<td>7.126</td>
</tr>
<tr>
<td>MOV Rs, Ri</td>
<td>A</td>
<td>B7</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>Rs → Ri</td>
<td>Rs: Dedicated Register</td>
<td>7.127</td>
</tr>
<tr>
<td>MOV Ri, Rs</td>
<td>A</td>
<td>B3</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>Ri → Rs</td>
<td>Rs: Dedicated Register</td>
<td>7.129</td>
</tr>
<tr>
<td>MOV PS, Ri</td>
<td>E</td>
<td>17-1</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>PS → Ri</td>
<td>PS: Program Status</td>
<td>7.128</td>
</tr>
<tr>
<td>MOV Ri, PS</td>
<td>E</td>
<td>07-1</td>
<td>c</td>
<td>CCCC</td>
<td>-</td>
<td>Ri → PS</td>
<td>PS: Program Status</td>
<td>7.130</td>
</tr>
</tbody>
</table>
### Table A.2-11 Non-delayed Branching Instructions (24 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FLAG</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP @Ri</td>
<td>E</td>
<td>97-0</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>Ri → PC</td>
<td></td>
<td>7.94</td>
</tr>
<tr>
<td>CALL label12</td>
<td>F</td>
<td>D0</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>PC+2 → RP, PC+2+exts(rel11 × 2) → PC</td>
<td></td>
<td>7.27</td>
</tr>
<tr>
<td>CALL @Ri</td>
<td>E</td>
<td>97-1</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>PC+2 → RP, Ri → PC</td>
<td></td>
<td>7.28</td>
</tr>
<tr>
<td>LCALL label21</td>
<td>I</td>
<td>07-2</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>PC+4 → RP, PC+4+exts(rel20 × 2) → PC</td>
<td></td>
<td>7.96</td>
</tr>
<tr>
<td>RET</td>
<td>E'</td>
<td>97-2</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>RP → PC</td>
<td></td>
<td>7.143</td>
</tr>
<tr>
<td>INT #u8</td>
<td>D</td>
<td>1F</td>
<td>1+3a</td>
<td>----</td>
<td>-</td>
<td>SSP-4 → SSP, PS → (SSP), SSP-4 → SSP, PC+2 → (SSP), 0 → CCR:S, (TBR+3FC-u8 × 4) → PC</td>
<td></td>
<td>7.92</td>
</tr>
<tr>
<td>INTE</td>
<td>E'</td>
<td>9F-3</td>
<td>1+3a</td>
<td>----</td>
<td>-</td>
<td>SSP → SSP, PS → (SSP), SSP → SSP, PC+2 → (SSP), 0 → CCR:S, 4 → ILM, (TBR+3D8) → PC</td>
<td></td>
<td>7.93</td>
</tr>
<tr>
<td>RETI</td>
<td>E'</td>
<td>97-3</td>
<td>1+2b</td>
<td>----</td>
<td>-</td>
<td>(SSP) → PC, SSP+4 → SSP, (SSP) → PS, SSP+4 → SSP</td>
<td></td>
<td>7.145</td>
</tr>
<tr>
<td>BNO label9</td>
<td>D</td>
<td>E1</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>No branch</td>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>BRA label9</td>
<td>D</td>
<td>E0</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>PC+2+exts(rel8 × 2) → PC</td>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>BEQ label9</td>
<td>D</td>
<td>E2</td>
<td>2/1</td>
<td>----</td>
<td>-</td>
<td>if (Z==1) then PC+2+exts(rel8 × 2) → PC</td>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>BNE label9</td>
<td>D</td>
<td>E3</td>
<td>2/1</td>
<td>----</td>
<td>-</td>
<td>if (Z==0) then PC+2+exts(rel8 × 2) → PC</td>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>BC label9</td>
<td>D</td>
<td>E4</td>
<td>2/1</td>
<td>----</td>
<td>-</td>
<td>if (C==1) then PC+2+exts(rel8 × 2) → PC</td>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>BNC label9</td>
<td>D</td>
<td>E5</td>
<td>2/1</td>
<td>----</td>
<td>-</td>
<td>if (C==0) then PC+2+exts(rel8 × 2) → PC</td>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>BN label9</td>
<td>D</td>
<td>E6</td>
<td>2/1</td>
<td>----</td>
<td>-</td>
<td>if (N==1) then PC+2+exts(rel8 × 2) → PC</td>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>BP label9</td>
<td>D</td>
<td>E7</td>
<td>2/1</td>
<td>----</td>
<td>-</td>
<td>if (N==0) then PC+2+exts(rel8 × 2) → PC</td>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>BV label9</td>
<td>D</td>
<td>E8</td>
<td>2/1</td>
<td>----</td>
<td>-</td>
<td>if (V==1) then PC+2+exts(rel8 × 2) → PC</td>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>BNV label9</td>
<td>D</td>
<td>E9</td>
<td>2/1</td>
<td>----</td>
<td>-</td>
<td>if (V==0) then PC+2+exts(rel8 × 2) → PC</td>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>BLT label9</td>
<td>D</td>
<td>EA</td>
<td>2/1</td>
<td>----</td>
<td>-</td>
<td>if (V ^ N==1) then PC+2+exts(rel8 × 2) → PC</td>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>BGE label9</td>
<td>D</td>
<td>EB</td>
<td>2/1</td>
<td>----</td>
<td>-</td>
<td>if (V ^ N==0) then PC+2+exts(rel8 × 2) → PC</td>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>BLE label9</td>
<td>D</td>
<td>EC</td>
<td>2/1</td>
<td>----</td>
<td>-</td>
<td>if ((V ^ N)</td>
<td>Z==1) then PC+2+exts(rel8 × 2) → PC</td>
<td></td>
</tr>
<tr>
<td>BGT label9</td>
<td>D</td>
<td>ED</td>
<td>2/1</td>
<td>----</td>
<td>-</td>
<td>if ((V ^ N)</td>
<td>Z==0) then PC+2+exts(rel8 × 2) → PC</td>
<td></td>
</tr>
<tr>
<td>BLS label9</td>
<td>D</td>
<td>EE</td>
<td>2/1</td>
<td>----</td>
<td>-</td>
<td>if (C or Z==1) then PC+2+exts(rel8 × 2) → PC</td>
<td></td>
<td>7.19</td>
</tr>
<tr>
<td>BHI label9</td>
<td>D</td>
<td>EF</td>
<td>2/1</td>
<td>----</td>
<td>-</td>
<td>if (C or Z==0) then PC+2+exts(rel8 × 2) → PC</td>
<td></td>
<td>7.19</td>
</tr>
</tbody>
</table>

- The value of "2/1" in CYC Column indicates 2 cycles if branching and 1 if not branching.
- It is necessary to set the Stack Flag (S) to "0" for RETI instruction execution.
The field rel8 in TYPE-D Instruction Format and the field rel11 in TYPE-F Format have the following relation to the values of label9, label12 in assembly notation.

\[
\text{rel8} = \frac{(\text{label9} - \text{PC} - 2)}{2} \\
\text{rel11} = \frac{(\text{label12} - \text{PC} - 2)}{2}
\]

The field rel20 in TYPE-I Instruction Format has the following relation to the values of label21 in assembly notation.

\[
\text{rel20} = \frac{(\text{label21} - \text{PC} - 4)}{2}
\]

Table A.2-12  Delayed Branching Instructions (21 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FLAG</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP: D @Ri</td>
<td>E</td>
<td>9F-0</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>Ri \rightarrow PC</td>
<td></td>
<td>7.95</td>
</tr>
<tr>
<td>CALL: D label12</td>
<td>F</td>
<td>D8</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>PC+4 \rightarrow RP, PC+4+exts(rel11 \times 2) \rightarrow PC</td>
<td></td>
<td>7.29</td>
</tr>
<tr>
<td>CALL: D @Ri</td>
<td>E</td>
<td>9F-1</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>PC+4 \rightarrow RP, Ri \rightarrow PC</td>
<td></td>
<td>7.30</td>
</tr>
<tr>
<td>LCALL: D label21</td>
<td>I</td>
<td>17-2</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>PC+6 \rightarrow RP, PC+4+exts(rel20 \times 2) \rightarrow PC</td>
<td></td>
<td>7.97</td>
</tr>
<tr>
<td>RET: D</td>
<td>E'</td>
<td>9F-2</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>RP \rightarrow PC</td>
<td></td>
<td>7.144</td>
</tr>
<tr>
<td>BNO: D label9</td>
<td>D</td>
<td>F1</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>No branch</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BRA: D label9</td>
<td>D</td>
<td>F0</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BEQ: D label9</td>
<td>D</td>
<td>F2</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>if (Z==1) then PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BNE: D label9</td>
<td>D</td>
<td>F3</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>if (Z==0) then PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BC: D label9</td>
<td>D</td>
<td>F4</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>if (C==1) then PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BNC: D label9</td>
<td>D</td>
<td>F5</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>if (C==0) then PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BN: D label9</td>
<td>D</td>
<td>F6</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>if (N==1) then PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BP: D label9</td>
<td>D</td>
<td>F7</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>if (N==0) then PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BV: D label9</td>
<td>D</td>
<td>F8</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>if (V==1) then PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BNV: D label9</td>
<td>D</td>
<td>F9</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>if (V==0) then PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BLT: D label9</td>
<td>D</td>
<td>FA</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>if (V \land N==1) then PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BGE: D label9</td>
<td>D</td>
<td>FB</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>if (V \land N==0) then PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BLE: D label9</td>
<td>D</td>
<td>FC</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>if (V \land N) \land Z==1 then PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BGT: D label9</td>
<td>D</td>
<td>FD</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>if (V \land N) \land Z==0 then PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BLS: D label9</td>
<td>D</td>
<td>FE</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>if (C or Z==1) then PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
<tr>
<td>BHI: D label9</td>
<td>D</td>
<td>FF</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>if (C or Z==0) then PC+2+exts(rel8 \times 2) \rightarrow PC</td>
<td></td>
<td>7.20</td>
</tr>
</tbody>
</table>
• Delayed Branching Instructions are branched after always executing the following Instruction (the Delay Slot).

• The field rel8 in TYPE-D instruction format and the field rel11 in TYPE-D format have the following relation to the values label9, label12 in assembly notation.

\[ \text{rel8} = (\text{label9-PC-2})/2 \]
\[ \text{rel11} = (\text{label12-PC-2})/2 \]

• The field rel20 in TYPE-I Instruction Format has the following relation to the values of label21 in assembly notation.

\[ \text{rel20} = (\text{label21-PC-4})/2 \]

**Table A.2-13 Direct Addressing Instructions (14 Instructions)**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FLAG NZVC</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMOV @dir10, R13</td>
<td>D</td>
<td>08</td>
<td>b</td>
<td>----</td>
<td>-</td>
<td>(dir8 × 4) → R13</td>
<td>Word</td>
<td>7.40</td>
</tr>
<tr>
<td>DMOV R13, @dir10</td>
<td>D</td>
<td>18</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td>R13 → (dir8 × 4)</td>
<td></td>
<td>7.41</td>
</tr>
<tr>
<td>DMOV @dir10, @R13+</td>
<td>D</td>
<td>0C</td>
<td>1+2a</td>
<td>----</td>
<td>-</td>
<td>(dir8 × 4) → (R13), R13+4 → (R13)</td>
<td></td>
<td>7.42</td>
</tr>
<tr>
<td>DMOV @R13+, @dir10</td>
<td>D</td>
<td>1C</td>
<td>1+2a</td>
<td>----</td>
<td>-</td>
<td>(R13) → (dir8 × 4), R13+4 → (R13)</td>
<td></td>
<td>7.43</td>
</tr>
<tr>
<td>DMOV @dir10, @-R15</td>
<td>D</td>
<td>0B</td>
<td>1+2a</td>
<td>----</td>
<td>-</td>
<td>R15-4 → (R15), (dir8 × 4) → (R15)</td>
<td></td>
<td>7.44</td>
</tr>
<tr>
<td>DMOV @R15+, @dir10</td>
<td>D</td>
<td>1B</td>
<td>1+2a</td>
<td>----</td>
<td>-</td>
<td>(R15) → (dir8 × 4), R15+4 → (R15)</td>
<td></td>
<td>7.45</td>
</tr>
<tr>
<td>DMOVH @dir9, R13</td>
<td>D</td>
<td>09</td>
<td>b</td>
<td>----</td>
<td>-</td>
<td>(dir8 × 2) → R13</td>
<td>Half-Word</td>
<td>7.50</td>
</tr>
<tr>
<td>DMOVH R13, @dir9</td>
<td>D</td>
<td>19</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td>R13 → (dir8 × 2)</td>
<td></td>
<td>7.51</td>
</tr>
<tr>
<td>DMOVH @dir9, @R13+</td>
<td>D</td>
<td>0D</td>
<td>1+2a</td>
<td>----</td>
<td>-</td>
<td>(dir8 × 2) → (R13), R13+2 → (R13)</td>
<td></td>
<td>7.52</td>
</tr>
<tr>
<td>DMOVH @R13+, @dir9</td>
<td>D</td>
<td>1D</td>
<td>1+2a</td>
<td>----</td>
<td>-</td>
<td>(R13) → (dir8 × 2), R13+2 → (R13)</td>
<td></td>
<td>7.53</td>
</tr>
<tr>
<td>DMOVB @dir8, R13</td>
<td>D</td>
<td>0A</td>
<td>b</td>
<td>----</td>
<td>-</td>
<td>(dir8) → R13</td>
<td>Byte</td>
<td>7.46</td>
</tr>
<tr>
<td>DMOVB R13, @dir8</td>
<td>D</td>
<td>1A</td>
<td>a</td>
<td>----</td>
<td>-</td>
<td>R13 → (dir8)</td>
<td></td>
<td>7.47</td>
</tr>
<tr>
<td>DMOVB @dir8, @R13+</td>
<td>D</td>
<td>0E</td>
<td>1+2a</td>
<td>----</td>
<td>-</td>
<td>(dir8) → (R13), R13+2 → (R13)</td>
<td></td>
<td>7.48</td>
</tr>
<tr>
<td>DMOVB @R13+, @dir8</td>
<td>D</td>
<td>1E</td>
<td>1+2a</td>
<td>----</td>
<td>-</td>
<td>(R13) → (dir8), R13+2 → (R13)</td>
<td></td>
<td>7.49</td>
</tr>
</tbody>
</table>

• The field dir8 in FORMAT_D Instruction format has the following relation to the values of dir8, dir9, dir10 in assembly notation.

\[ \text{dir8} = \text{dir8} \]
\[ \text{dir8} = \text{dir9} >> 1 \]
\[ \text{dir8} = \text{dir10} >> 2 \]
Table A.2-14 Bit Search Instructions (3 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FLAG</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRCH0 Ri</td>
<td>E</td>
<td>97-C</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>search_zero(Ri) → Ri</td>
<td>Searches first 0 Bit</td>
<td>7.110</td>
</tr>
<tr>
<td>SRCH1 Ri</td>
<td>E</td>
<td>97-D</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>search_one(Ri) → Ri</td>
<td>Searches first 1 Bit</td>
<td>7.111</td>
</tr>
<tr>
<td>SRCHC Ri</td>
<td>E</td>
<td>97-E</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>search_change(Ri) → Ri</td>
<td>Searches first change</td>
<td>7.112</td>
</tr>
</tbody>
</table>

Table A.2-15 Other Instructions (16 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FLAG</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>E'</td>
<td>9F-A</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>No change</td>
<td></td>
<td>7.137</td>
</tr>
<tr>
<td>ANDCCR #u8</td>
<td>D</td>
<td>83</td>
<td>1</td>
<td>CCCC</td>
<td>-</td>
<td>CCR &amp; u8 → CCR</td>
<td></td>
<td>7.12</td>
</tr>
<tr>
<td>ORCCR #u8</td>
<td>D</td>
<td>93</td>
<td>1</td>
<td>CCCC</td>
<td>-</td>
<td>CCR</td>
<td>u8 → CCR</td>
<td></td>
</tr>
<tr>
<td>STILM #u8</td>
<td>D</td>
<td>87</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>u8 → ILM</td>
<td>Sets ILM immediate value</td>
<td>7.126</td>
</tr>
<tr>
<td>ADDSP #10</td>
<td>D</td>
<td>A3</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>R15+u8 ×4 → R15</td>
<td></td>
<td>7.8</td>
</tr>
<tr>
<td>EXTSB Ri</td>
<td>E</td>
<td>97-8</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>exts(Ri[7:0]) → Ri</td>
<td>Sign extension 8 → 32</td>
<td>7.59</td>
</tr>
<tr>
<td>EXTUB Ri</td>
<td>E</td>
<td>97-9</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>extu(Ri[7:0]) → Ri</td>
<td>Zero extension 8 → 32</td>
<td>7.61</td>
</tr>
<tr>
<td>EXTSH Ri</td>
<td>E</td>
<td>97-A</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>exts(Ri[15:0]) → Ri</td>
<td>Sign extension 16 → 32</td>
<td>7.60</td>
</tr>
<tr>
<td>EXTUH Ri</td>
<td>E</td>
<td>97-B</td>
<td>1</td>
<td>----</td>
<td>-</td>
<td>extu(Ri[15:0]) → Ri</td>
<td>Zero extension 16 → 32</td>
<td>7.62</td>
</tr>
<tr>
<td>LDM0 (reglist)</td>
<td>D</td>
<td>8C</td>
<td>*1</td>
<td>----</td>
<td>-</td>
<td>for Ri of reglist (R15) → Ri, R15+4 → R15</td>
<td>Load Multiple R0 to R7</td>
<td>7.109</td>
</tr>
<tr>
<td>LDM1 (reglist)</td>
<td>D</td>
<td>8D</td>
<td>*1</td>
<td>----</td>
<td>-</td>
<td>for Ri of reglist (R15) → Ri, R15+4 → R15</td>
<td>Load Multiple R8 to R15</td>
<td>7.110</td>
</tr>
<tr>
<td>STM0 (reglist)</td>
<td>D</td>
<td>8E</td>
<td>*2</td>
<td>----</td>
<td>-</td>
<td>for Ri of reglist R15-4 → R15, Ri → (R15)</td>
<td>Store multiple R0 to R7</td>
<td>7.127</td>
</tr>
<tr>
<td>STM1 (reglist)</td>
<td>D</td>
<td>8F</td>
<td>*2</td>
<td>----</td>
<td>-</td>
<td>for Ri of reglist R15-4 → R15, Ri → (R15)</td>
<td>Store multiple R8 to R15</td>
<td>7.128</td>
</tr>
<tr>
<td>ENTER #u10</td>
<td>D</td>
<td>0F</td>
<td>1+a</td>
<td>----</td>
<td>-</td>
<td>R14 → (R15-4), R15-4 → R14, R15-extu(u8 ×4) → R15</td>
<td>Function entry processing</td>
<td>7.54</td>
</tr>
<tr>
<td>LEAVE</td>
<td>E'</td>
<td>9F-9</td>
<td>b</td>
<td>----</td>
<td>-</td>
<td>R14+4 → R15, (R15-4) → R14</td>
<td>Function exit processing</td>
<td>7.85</td>
</tr>
<tr>
<td>XCHB @Rj, Ri</td>
<td>A</td>
<td>8A</td>
<td>2a</td>
<td>----</td>
<td>-</td>
<td>Ri → TEMP, extu((Rj)) → Ri, TEMP → (Rj)</td>
<td>Byte data for semaphore processing</td>
<td>7.132</td>
</tr>
</tbody>
</table>

*1: The number of execution cycles for LDM0(reglist) and LDM1(reglist) is b × n cycles when "n" is the number of registers designated.

*2: The number of execution cycles for STM0(reglist) and STM1(reglist) is a × n when "n" is the number of registers designated.

- In the ADD SP Instruction, the field s8 in TYPR-D Instruction Format has the following relation to the value of s10 in assembly notation.
  
s8 = s10 >> 2

- In the ENTER Instruction, the field u8 in TYPR-D Instruction Format has the following relation to the value of u10 in assembly notation.
  
u8 = u10 >> 2
APPENDIX
APPENDIX A  Instruction Lists

FR81 Family

Table A.2-16  FPU Memory Load Instructions (7 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FCC</th>
<th>ELGU</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD  @Rj, FRi</td>
<td>K</td>
<td>07-C</td>
<td>a</td>
<td>----</td>
<td></td>
<td></td>
<td>(Rj) → FRi</td>
<td>Word</td>
<td>7.70</td>
</tr>
<tr>
<td>FLD  @(R13, Rj), FRi</td>
<td>K</td>
<td>07-E</td>
<td>a</td>
<td>----</td>
<td></td>
<td></td>
<td>(R13+Rj) → FRi</td>
<td>Word</td>
<td>7.71</td>
</tr>
<tr>
<td>FLD  @(R14, disp16), FRi</td>
<td>L</td>
<td>07-D0</td>
<td>a</td>
<td>----</td>
<td></td>
<td></td>
<td>(R14+o14×4) → FRi</td>
<td>Word</td>
<td>7.72</td>
</tr>
<tr>
<td>FLD  @(R15, udisp16), FRi</td>
<td>L</td>
<td>07-D4</td>
<td>a</td>
<td>----</td>
<td></td>
<td></td>
<td>(R15+u14×4) → FRi</td>
<td>Word</td>
<td>7.73</td>
</tr>
<tr>
<td>FLD  @R15+, FRi</td>
<td>L</td>
<td>07-D8</td>
<td>a</td>
<td>----</td>
<td></td>
<td></td>
<td>(R15) → FRi</td>
<td>Word</td>
<td>7.74</td>
</tr>
<tr>
<td>FLD  @(BP, udisp18), FRi</td>
<td>J</td>
<td>07-7</td>
<td>a</td>
<td>----</td>
<td></td>
<td></td>
<td>(BP+u16×4) → FRi</td>
<td>Word</td>
<td>7.75</td>
</tr>
<tr>
<td>FLDM (frlist)</td>
<td>N</td>
<td>07-DC</td>
<td>*1</td>
<td>----</td>
<td></td>
<td></td>
<td>Load Multiple</td>
<td>FRi of frlist</td>
<td>7.76</td>
</tr>
</tbody>
</table>

*1: The number of execution cycles for FLDM instruction is a × n when "n" is the number of registers designated.

- The field o14 and u14 in TYPE-L instruction format have the following relation to the values disp16, udisp16 in assembly notation.
  
  o14 = disp16 >> 2
  
  u14 = udisp16 >> 2

- The field u16 in TYPE-J instruction format has the following relation to the value udisp18 in assembly notation.
  
  u16 = udisp18 >> 2

Table A.2-17  FPU Memory Store Instructions (7 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FCC</th>
<th>ELGU</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FST  FRi, @Rj</td>
<td>K</td>
<td>17-C</td>
<td>a</td>
<td>----</td>
<td></td>
<td></td>
<td>FRi → (Rj)</td>
<td>Word</td>
<td>7.83</td>
</tr>
<tr>
<td>FST  FRi, @(R13, Rj)</td>
<td>K</td>
<td>17-E</td>
<td>a</td>
<td>----</td>
<td></td>
<td></td>
<td>FRi → (R13+Rj)</td>
<td>Word</td>
<td>7.84</td>
</tr>
<tr>
<td>FST  FRi, @(R14, disp16)</td>
<td>L</td>
<td>17-D0</td>
<td>a</td>
<td>----</td>
<td></td>
<td></td>
<td>FRi → (R14+o14×4)</td>
<td>Word</td>
<td>7.85</td>
</tr>
<tr>
<td>FST  FRi, @(R15, udisp16)</td>
<td>L</td>
<td>17-D4</td>
<td>a</td>
<td>----</td>
<td></td>
<td></td>
<td>FRi → (R15+u14×4)</td>
<td>Word</td>
<td>7.86</td>
</tr>
<tr>
<td>FST  FRi, @-R15</td>
<td>L</td>
<td>17-D8</td>
<td>a</td>
<td>----</td>
<td></td>
<td></td>
<td>R15-4 → R15</td>
<td>Word</td>
<td>7.87</td>
</tr>
<tr>
<td>FST  FRi, @(BP, udisp18)</td>
<td>J</td>
<td>17-7</td>
<td>a</td>
<td>----</td>
<td></td>
<td></td>
<td>FRi → (BP+u16×4)</td>
<td>Word</td>
<td>7.88</td>
</tr>
<tr>
<td>FSTM (frlist)</td>
<td>N</td>
<td>17-DC</td>
<td>*1</td>
<td>----</td>
<td></td>
<td></td>
<td>Store Multiple FR0 to FR15</td>
<td>Word</td>
<td>7.89</td>
</tr>
</tbody>
</table>

*1: The number of execution cycles for FSTM instruction is a × n when "n" is the number of registers designated.

- The field o14 and u14 in TYPE-L instruction format have the following relation to the values disp16 and udisp16 in assembly notation.
  
  o14 = disp16 >> 2
  
  u14 = udisp16 >> 2

- The field u16 in TYPE-J instruction format has the following relation to the value udisp18 in assembly notation.
  
  u16 = udisp18 >> 2
## Table A.2-18  FPU Single-Precision Floating Point Calculation Instruction (12 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FCC</th>
<th>ELGU</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADDs</td>
<td>FRk, FRj, FRi</td>
<td>M</td>
<td>07-A0</td>
<td>1</td>
<td>----</td>
<td>----</td>
<td>FRk + FRj → FRi</td>
<td></td>
<td>7.64</td>
</tr>
<tr>
<td>FSUBs</td>
<td>FRk, FRj, FRi</td>
<td>M</td>
<td>07-A2</td>
<td>1</td>
<td>----</td>
<td>----</td>
<td>FRk - FRj → FRi</td>
<td></td>
<td>7.91</td>
</tr>
<tr>
<td>FCMPs</td>
<td>FRk, FRj</td>
<td>M</td>
<td>07-A4</td>
<td>1</td>
<td>CCCC</td>
<td>----</td>
<td>FRk - FRj</td>
<td></td>
<td>7.67</td>
</tr>
<tr>
<td>FMULs</td>
<td>FRk, FRj, FRi</td>
<td>M</td>
<td>07-A7</td>
<td>1</td>
<td>----</td>
<td>----</td>
<td>FRk × FRj → FRi</td>
<td></td>
<td>7.80</td>
</tr>
<tr>
<td>FDIVs</td>
<td>FRk, FRj, FRi</td>
<td>M</td>
<td>07-AA</td>
<td>9</td>
<td>----</td>
<td>----</td>
<td>FRk / FRj → FRi</td>
<td></td>
<td>7.68</td>
</tr>
<tr>
<td>FNEGs</td>
<td>FRj, FRi</td>
<td>M</td>
<td>07-AF</td>
<td>1</td>
<td>----</td>
<td>----</td>
<td>FRj × -1 → FRi</td>
<td></td>
<td>7.81</td>
</tr>
<tr>
<td>FABSs</td>
<td>FRj, FRi</td>
<td>M</td>
<td>07-AF</td>
<td>1</td>
<td>----</td>
<td>----</td>
<td></td>
<td></td>
<td>7.63</td>
</tr>
<tr>
<td>FMADDs</td>
<td>FRk, FRj, FRi</td>
<td>M</td>
<td>07-A5</td>
<td>4</td>
<td>----</td>
<td>----</td>
<td>FRk × FRj + FRi → FRi</td>
<td></td>
<td>7.77</td>
</tr>
<tr>
<td>FMSUBs</td>
<td>FRk, FRj, FRi</td>
<td>M</td>
<td>07-A6</td>
<td>4</td>
<td>----</td>
<td>----</td>
<td>FRk × FRj - FRi → FRi</td>
<td></td>
<td>7.79</td>
</tr>
<tr>
<td>FSQRTs</td>
<td>FRj, FRi</td>
<td>M</td>
<td>07-AB</td>
<td>14</td>
<td>----</td>
<td>----</td>
<td>√FRj → FRi</td>
<td></td>
<td>7.82</td>
</tr>
<tr>
<td>FiTOs</td>
<td>FRj, FRi</td>
<td>M</td>
<td>07-A8</td>
<td>1</td>
<td>----</td>
<td>----</td>
<td>(float) FRj → FRi</td>
<td></td>
<td>7.69</td>
</tr>
<tr>
<td>FsTOi</td>
<td>FRj, FRi</td>
<td>M</td>
<td>07-A9</td>
<td>1</td>
<td>----</td>
<td>----</td>
<td>(int) FRj → FRi</td>
<td></td>
<td>7.90</td>
</tr>
</tbody>
</table>

## Table A.2-19  FPU Inter-Register Transfer Instruction (3 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FCC</th>
<th>ELGU</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMOVs</td>
<td>FRj, FRi</td>
<td>M</td>
<td>07-AE</td>
<td>1</td>
<td>----</td>
<td>----</td>
<td>FRj → FRi</td>
<td></td>
<td>7.78</td>
</tr>
<tr>
<td>MOV</td>
<td>Rj, FRi</td>
<td>K</td>
<td>07-3</td>
<td>1</td>
<td>----</td>
<td>----</td>
<td>Rj → FRi</td>
<td></td>
<td>7.131</td>
</tr>
<tr>
<td>MOV</td>
<td>FRj, Ri</td>
<td>K</td>
<td>17-3</td>
<td>1</td>
<td>----</td>
<td>----</td>
<td>FRj → Ri</td>
<td></td>
<td>7.132</td>
</tr>
</tbody>
</table>
### Table A.2-20  FPU Branching Instruction without Delay (16 Instructions)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>FCC</th>
<th>ELGU</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBN</td>
<td>N</td>
<td>07-F0</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>No branch</td>
<td></td>
<td>7.65</td>
</tr>
<tr>
<td>FBA label17</td>
<td>N</td>
<td>07-FF</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>PC+4+exts(rel16 × 2) → PC</td>
<td></td>
<td>7.65</td>
<td></td>
</tr>
<tr>
<td>FBNE label17</td>
<td>N</td>
<td>07-F7</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>if (L</td>
<td></td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>FBE label17</td>
<td>N</td>
<td>07-F8</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>if (E) then PC+4+exts(rel16 × 2) → PC</td>
<td></td>
<td>7.65</td>
<td></td>
</tr>
<tr>
<td>FBLG label17</td>
<td>N</td>
<td>07-F6</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>if (L</td>
<td></td>
<td>G) then PC+4+exts(rel16 × 2) → PC</td>
<td></td>
</tr>
<tr>
<td>FBUE label17</td>
<td>N</td>
<td>07-F9</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>if (E</td>
<td></td>
<td>U) then PC+4+exts(rel16 × 2) → PC</td>
<td></td>
</tr>
<tr>
<td>FBUL label17</td>
<td>N</td>
<td>07-F5</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>if (L</td>
<td></td>
<td>U) then PC+4+exts(rel16 × 2) → PC</td>
<td></td>
</tr>
<tr>
<td>FBGE label17</td>
<td>N</td>
<td>07-FA</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>if (G</td>
<td></td>
<td>E) then PC+4+exts(rel16 × 2) → PC</td>
<td></td>
</tr>
<tr>
<td>FBL label17</td>
<td>N</td>
<td>07-F4</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>if (L) then PC+4+exts(rel16 × 2) → PC</td>
<td></td>
<td>7.65</td>
<td></td>
</tr>
<tr>
<td>FBUGE label17</td>
<td>N</td>
<td>07-FB</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>if (U</td>
<td></td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>FBUG label17</td>
<td>N</td>
<td>07-F3</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>if (U</td>
<td></td>
<td>G) then PC+4+exts(rel16 × 2) → PC</td>
<td></td>
</tr>
<tr>
<td>FBLE label17</td>
<td>N</td>
<td>07-FC</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>if (L</td>
<td></td>
<td>E) then PC+4+exts(rel16 × 2) → PC</td>
<td></td>
</tr>
<tr>
<td>FBG label17</td>
<td>N</td>
<td>07-F2</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>if (G) then PC+4+exts(rel16 × 2) → PC</td>
<td></td>
<td>7.65</td>
<td></td>
</tr>
<tr>
<td>FBULE label17</td>
<td>N</td>
<td>07-FD</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>if (E</td>
<td></td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>FBU label17</td>
<td>N</td>
<td>07-F1</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>if (U) then PC+4+exts(rel16 × 2) → PC</td>
<td></td>
<td>7.65</td>
<td></td>
</tr>
<tr>
<td>FBO label17</td>
<td>N</td>
<td>07-FE</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>if (E</td>
<td></td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>

- The field rel16 in TYPE-N instruction format has the following relation to the value label17 in assembly notation.

\[
    \text{rel16} = (\text{label17} - \text{PC} - 4) / 2
\]
### FR81 Family

#### APPENDIX

#### APPENDIX A Instruction Lists

**Table A.2-21 FPU Branching Instruction with Delay (16 Instructions)**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>OP</th>
<th>CYC</th>
<th>ECC</th>
<th>EELG</th>
<th>RMW</th>
<th>Operation</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBN:D</td>
<td>N</td>
<td>17-F0</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>No branch</td>
<td></td>
<td>7.66</td>
</tr>
<tr>
<td>FBA:D label17</td>
<td>N</td>
<td>17-FF</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>PC+4+exts(rel16 × 2) → PC</td>
<td></td>
<td>7.66</td>
</tr>
<tr>
<td>FBN:E label17</td>
<td>N</td>
<td>17-F7</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>if (L</td>
<td></td>
<td>G</td>
</tr>
<tr>
<td>FBE:D label17</td>
<td>N</td>
<td>17-F8</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>if (E) then PC+4+exts(rel16 × 2) → PC</td>
<td>7.66</td>
<td></td>
</tr>
<tr>
<td>FBLG:D label17</td>
<td>N</td>
<td>17-F6</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>if (L</td>
<td></td>
<td>G ) then PC+4+exts(rel16 × 2) → PC</td>
</tr>
<tr>
<td>FBU:E label17</td>
<td>N</td>
<td>17-F9</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>if (E</td>
<td></td>
<td>U ) then PC+4+exts(rel16 × 2) → PC</td>
</tr>
<tr>
<td>FBL:D label17</td>
<td>N</td>
<td>17-F5</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>if (L</td>
<td></td>
<td>U ) then PC+4+exts(rel16 × 2) → PC</td>
</tr>
<tr>
<td>FBUG:E label17</td>
<td>N</td>
<td>17-FA</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>if (G</td>
<td></td>
<td>E ) then PC+4+exts(rel16 × 2) → PC</td>
</tr>
<tr>
<td>FBL:E label17</td>
<td>N</td>
<td>17-F4</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>if (L) then PC+4+exts(rel16 × 2) → PC</td>
<td>7.66</td>
<td></td>
</tr>
<tr>
<td>FBUG:E label17</td>
<td>N</td>
<td>17-FB</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>if (U</td>
<td></td>
<td>G</td>
</tr>
<tr>
<td>FBUG:D label17</td>
<td>N</td>
<td>17-F3</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>if (U</td>
<td></td>
<td>G ) then PC+4+exts(rel16 × 2) → PC</td>
</tr>
<tr>
<td>FBLE:D label17</td>
<td>N</td>
<td>17-FC</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>if (L</td>
<td></td>
<td>E ) then PC+4+exts(rel16 × 2) → PC</td>
</tr>
<tr>
<td>FBGE:D label17</td>
<td>N</td>
<td>17-F2</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>if (G) then PC+4+exts(rel16 × 2) → PC</td>
<td>7.66</td>
<td></td>
</tr>
<tr>
<td>FBUG:E label17</td>
<td>N</td>
<td>17-FD</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>if (E</td>
<td></td>
<td>L</td>
</tr>
<tr>
<td>FBU:D label17</td>
<td>N</td>
<td>17-F1</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>if (U) then PC+4+exts(rel16 × 2) → PC</td>
<td>7.66</td>
<td></td>
</tr>
<tr>
<td>FBO:D label17</td>
<td>N</td>
<td>17-FE</td>
<td>2</td>
<td>----</td>
<td>-</td>
<td>-</td>
<td>if (E</td>
<td></td>
<td>L</td>
</tr>
</tbody>
</table>

- Delayed Branching Instructions are branched after always executing the following Instruction (the Delay Slot).
- The field rel16 in TYPE-N instruction format has the following relation to the value label17 in assembly notation.

\[
rel16 = (label17 - PC - 4) / 2
\]
### A.3 List of Instructions that can be positioned in the Delay Slot

This section shows the Instructions List that can be positioned in the delay slot of Delay Branching Instruction.

- **Add/Subtract Instructions**
  - ADD Rj, Ri
  - ADD #14, Ri
  - ADD2 #i4, Ri
  - ADDC Rj, Ri
  - ADDN Rj, Ri
  - ADDN #i4, Ri
  - ADDN2 #i4, Ri
  - SUB Rj, Ri
  - SUBC Rj, Ri
  - SUBN Rj, Ri

- **Compare Calculation Instructions**
  - CMP Rj, Ri
  - CMP #i4, Ri
  - CMP2 #i4, Ri

- **Logical Calculation Instructions**
  - AND Rj, Ri
  - OR Rj, Ri
  - EOR Rj, Ri

- **Multiply/Divide Instructions**
  - DIV0S Ri
  - DIV0U Ri
  - DIV1 Ri
  - DIV2 Ri
  - DIV3
  - DIV4S

- **Shift Instructions**
  - LSL Rj, Ri
  - LSL #u4, Ri
  - LSL2 #u4, Ri
  - LSR Rj, Ri
  - LSR #u4, Ri
  - LSR2 #u4, Ri
  - ASR Rj, Ri
  - ASR #u4, Ri
  - ASR2 #u4, Ri

- **Immediate Data Transfer Instructions**
  - LDI:8 #i8, Ri

- **Memory Load Instructions**
  - LD @Rj, Ri
  - LD @(R13, Rj), Ri
  - LD @(R14, disp10), Ri
  - LD @(R15, udisp6), Ri
  - LD @R15+, Ri
  - LD @R15+, Rs
## APPENDIX A    Instruction Lists

### Memory Store Instructions

- **ST** Ri, @Rj
- **ST** Ri, @(R13, Rj)
- **ST** Ri, @(R14, disp10)
- **ST** Ri, @(-R15)
- **ST** Ps, @(-R15)
- **STH** Ri, @Rj
- **STH** Ri, @(R13, Rj)
- **STH** Ri, @(R14, disp9)
- **STB** Ri, @Rj
- **STB** Ri, @(R13, Rj)
- **STB** Ri, @(R14, disp8)

### Inter-Register Transfer Instructions

- **MOV** Rj, Ri
- **MOV** Rs, Ri
- **MOV** Ri, Rs
- **MOV** Ps, Ri
- **MOV** Ri, Ps

### Direct Addressing Instructions

- **DMOV** @dir10, R13
- **DMOV** R13, @dir10
- **DMOVH** @dir9, R13
- **DMOVH** R13, @dir9
- **DMOVB** @dir8, R13
- **DMOVB** R13, @dir8

### Bit Search Instructions

- **SRCH0** Ri
- **SRCH1** Ri
- **SRCHC** Ri

### Other Instructions

- **NOP**
- **ANDCCR** #u8
- **ORCCR** #u8
- **STILM** #u8
- **ADDSP** #s10
- **EXTSB** Ri
- **ExtUB** Ri
- **EXTSH** Ri
- **EXTUH** Ri
- **LEAVE**
APPENDIX B Instruction Maps

It includes instruction maps of FR81 Family CPU.

B.1 Instruction Maps
B.2 Extension Instruction Maps
The following shows an instruction map when the operation code consists of 8 or less bits.

Figure B.1-1 illustrates in tabular form 8 bit operation codes (OP) for each instruction. Instructions where operation code (OP) is less than 8 bit, they have been converted into 8 bit by packing them on MSB side.
The following shows an instruction map when the operation code consists of 12 or more bits.

Instructions with a 12-bit operation code (OP), which is divided into 8 higher bits and 4 lower bits are shown in Table B.2-1.

### Table B.2-1 Instruction Map with 12-Bit Operation Code

<table>
<thead>
<tr>
<th>Higher 8 bits</th>
<th>07</th>
<th>17</th>
<th>97</th>
<th>9F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LD @R15+,Ri</td>
<td>ST Ri, @-R15</td>
<td>JMP @Ri</td>
<td>JMP:D @Ri</td>
</tr>
<tr>
<td>1</td>
<td>MOV Ri,PS</td>
<td>MOV PS,Ri</td>
<td>CALL @Ri</td>
<td>CALL:D @Ri</td>
</tr>
<tr>
<td>2</td>
<td>LCALL label21</td>
<td>LCALL:D label21</td>
<td>RET</td>
<td>RET:D</td>
</tr>
<tr>
<td>3</td>
<td>MOV Rj, FRi</td>
<td>MOV FRi,Rj</td>
<td>RETI</td>
<td>INTE</td>
</tr>
<tr>
<td>4</td>
<td>LD @(BP, udisp18), Ri</td>
<td>ST Ri, @(BP, udisp18)</td>
<td>DIV0S Ri</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>LDUH @(BP, udisp17), Ri</td>
<td>STH Ri, @(BP, udisp17)</td>
<td>DIV0U Ri</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>LDUB @(BP, udisp16), Ri</td>
<td>STB Ri, @(BP, udisp16)</td>
<td>DIV1 Ri</td>
<td>DIV3</td>
</tr>
<tr>
<td>7</td>
<td>FLD @(BP, udisp18), FRi</td>
<td>FST FRi, @(BP, udisp18)</td>
<td>DIV2 Ri</td>
<td>DIV4S</td>
</tr>
<tr>
<td>8</td>
<td>LD @R15+,Rs</td>
<td>ST Rs, @-R15</td>
<td>EXTSB Ri</td>
<td>LDI:32 #i32,Ri</td>
</tr>
<tr>
<td>9</td>
<td>LD @R15+,PS</td>
<td>ST PS, @-R15</td>
<td>EXTUB Ri</td>
<td>LEAVE</td>
</tr>
<tr>
<td>A</td>
<td>Refer to Table B.2-2</td>
<td>Refer to Table B.2-2</td>
<td>EXTSBH Ri</td>
<td>NOP</td>
</tr>
<tr>
<td>B</td>
<td>-</td>
<td>-</td>
<td>EXTUBH Ri</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>FLD @Rj, FRi</td>
<td>FST FRi, @Rj</td>
<td>-</td>
<td>SRCH0</td>
</tr>
<tr>
<td>D</td>
<td>Refer to Table B.2-2</td>
<td>Refer to Table B.2-2</td>
<td>-</td>
<td>SRCH1</td>
</tr>
<tr>
<td>E</td>
<td>FLD @(R13, Rj), FRi</td>
<td>FST FRi, @(R13, Rj)</td>
<td>-</td>
<td>SRCHC</td>
</tr>
<tr>
<td>F</td>
<td>Refer to Table B.2-3</td>
<td>Refer to Table B.2-3</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

---

*: Undefined
Instructions with 16-bit and 14-bit operation codes (OP), each of which is divided into 8 higher bits and 8 lower bits are shown in Tables Table B.2-2 and Table B.2-3. An instruction with a 14-bit operation code is padded to the MSB side to convert the 14-bit operation code to a 16-bit operation code.

**Table B.2-2 Instruction Map with 16-Bit/14-Bit Operation Code**

<table>
<thead>
<tr>
<th>Higher 8 bits</th>
<th>07</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>FADDs FRK, FRj, FRi</td>
<td>-</td>
</tr>
<tr>
<td>A1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A2</td>
<td>FSUBs FRK, FRj, FRi</td>
<td>-</td>
</tr>
<tr>
<td>A3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A4</td>
<td>FCMPs FRk, FRj</td>
<td>-</td>
</tr>
<tr>
<td>A5</td>
<td>FMADDs FRk, FRj, FRi</td>
<td>-</td>
</tr>
<tr>
<td>A6</td>
<td>FMSUBs FRk, FRj, FRi</td>
<td>-</td>
</tr>
<tr>
<td>A7</td>
<td>FMULs FRK, FRj, FRi</td>
<td>-</td>
</tr>
<tr>
<td>A8</td>
<td>FiTOs FRj, FRi</td>
<td>-</td>
</tr>
<tr>
<td>A9</td>
<td>FsTOi FRj, FRi</td>
<td>-</td>
</tr>
<tr>
<td>AA</td>
<td>FDIVs FRK, FRj, FRi</td>
<td>-</td>
</tr>
<tr>
<td>AB</td>
<td>FSQRTs FRk, FRj</td>
<td>-</td>
</tr>
<tr>
<td>AC</td>
<td>FABSs FRj, FRi</td>
<td>-</td>
</tr>
<tr>
<td>AD</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AE</td>
<td>FMOVs FRj, FRi</td>
<td>-</td>
</tr>
<tr>
<td>AF</td>
<td>FNEGs FRj, FRi</td>
<td>-</td>
</tr>
<tr>
<td>D0*</td>
<td>FLD @(R14, disp16), FRi</td>
<td>FST FRi, @(R14, disp16)</td>
</tr>
<tr>
<td>D4*</td>
<td>FLD @(R15, udisp16), FRi</td>
<td>FST FRi, @(R15, udisp16)</td>
</tr>
<tr>
<td>D8*</td>
<td>FLD @R15+, FRi</td>
<td>FST FRi, @(R15, udisp16)</td>
</tr>
<tr>
<td>DC*</td>
<td>FLDM (frlist)</td>
<td>FSTM (frlist)</td>
</tr>
</tbody>
</table>

*: Instruction of 14-Bit Operation Code
-: Undefined
<table>
<thead>
<tr>
<th>Higher 8 bits</th>
<th>Lower 8 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>F0</strong> FBN</td>
<td>FBN:D</td>
</tr>
<tr>
<td><strong>F1</strong> FBU label17</td>
<td>FBU:D label17</td>
</tr>
<tr>
<td><strong>F2</strong> FBG label17</td>
<td>FBG:D label17</td>
</tr>
<tr>
<td><strong>F3</strong> FBUG label17</td>
<td>FBUG:D label17</td>
</tr>
<tr>
<td><strong>F4</strong> FBL label17</td>
<td>FBL:D label17</td>
</tr>
<tr>
<td><strong>F5</strong> FBUL label17</td>
<td>FBUL:D label17</td>
</tr>
<tr>
<td><strong>F6</strong> FBLG label17</td>
<td>FBLG:D label17</td>
</tr>
<tr>
<td><strong>F7</strong> FBNE label17</td>
<td>FBNE:D label17</td>
</tr>
<tr>
<td><strong>F8</strong> FBE label17</td>
<td>FBE:D label17</td>
</tr>
<tr>
<td><strong>F9</strong> FBUE label17</td>
<td>FBUE:D label17</td>
</tr>
<tr>
<td><strong>FA</strong> FBGE label17</td>
<td>FBGE:D label17</td>
</tr>
<tr>
<td><strong>FB</strong> FBUGE label17</td>
<td>FBUGE:D label17</td>
</tr>
<tr>
<td><strong>FC</strong> FBLE label17</td>
<td>FBLE:D label17</td>
</tr>
<tr>
<td><strong>FD</strong> FBULE label17</td>
<td>FBULE:D label17</td>
</tr>
<tr>
<td><strong>FE</strong> FBO label17</td>
<td>FBO:D label17</td>
</tr>
<tr>
<td><strong>FF</strong> FBA label17</td>
<td>FBA:D label17</td>
</tr>
</tbody>
</table>
C.1 Conformity with IEEE754-1985 Standard

The FR81 Family CPU conforms to the IEEE754-1985 standard (hereinafter referred to as "IEEE754"), excluding the following.

1. Overflow
   IEEE754 defines that the biased (+192 for single-precision) exponent part is used as the result if overflow occurs; however, this architecture does not provide for certain cases such as the addition of the biased value.

2. Underflow
   IEEE754 defines that the biased (-192 for single-precision) exponent part is used as the result if underflow occurs; however, this architecture does not provide for certain cases such as the reduction of the biased value.
   When the result is an unnormalized number, the result is flushed to zero.

3. Unnormalized number
   IEEE754 defines an unnormalized number to prevent the result from being flushed rapidly to zero; however, the unnormalized number is not supported in this architecture. If an unnormalized number is input when the unnormalized number input exception is prohibited, its numeric value is assumed to be zero for calculation purposes. This is also applied when the calculation result is an unnormalized number; therefore, the result is set to zero.

4. QNaN
   The QNaN output pattern is fixed to 7FFFFFFFFH, excluding the move, sign inversion, and absolute value instructions.

5. Unsupported instructions
   This architecture does not support the following instructions.
   • Remainder
   • Integer rounding (Round Floating-Point Number to Integer Value)
   • Conversion between binary and decimal numbers

6. Floating point calculation exception
   This exception occurs when the sufficient calculation result is not obtained for IEEE754. This architecture provides six exceptions in total: five exceptions (Inexact, Underflow, Overflow, Division by Zero, and Invalid Calculation), which are defined in IEEE754, as well as one exception (unnormalized number input).

In IEEE754, if a floating point calculation exception occurs, the defined operation is carried out to generate a trap. In this architecture, it is provided as an exception; therefore, no data is written to the destination when a floating point calculation exception occurs.
C.2 FPU Exceptions

FPU exceptions are classified into six types: five exceptions (Inexact, Underflow, Overflow, Division by Zero, and Invalid Calculation), which are defined in IEEE754, and an exception that is generated when an unnormalized number is input. Whether or not to generate those calculation exceptions can be specified using the FPU control register (FCR.EEF). The result output upon the exception conditions being satisfied varies depending on the FCR.EEF setting.

If an exception is not permitted, the result of each calculation is output so that the requested result is obtained when calculation runs on even if an exception occurs, or so that it can be recognized from the result that the calculation is invalid. When an exception is permitted, if the significant calculation result cannot be obtained due to an exception factor, the calculation result is not written; otherwise, it is written.

1. Invalid calculation exception (Invalid Operation)
   This exception occurs when calculation cannot be carried out properly because the specified operand is invalid for the calculation. In concrete terms, this exception occurs when:
   - SNaN has been input;
   - the result is in infinite format (\(\infty - \infty\), \(0 \times \infty\), \(0/0\), \(\infty/\infty\), \(\sqrt{-1}\) etc.)
   - the conversion source value is not indicated in the conversion destination format using a conversion instruction.

   If this exception occurs, the following operations are carried out.
   
   [FCR:EEF:V=1]
   Writing to the floating point register or FCR:FCC is prohibited.
   The FCR:CEF.V flag is set to generate an FPU exception.
   
   [FCR:EEF:V=0]
   QNaN (7FFFFFFFH) is stored in the floating point register for instructions other than conversion or comparison instructions.
   For the conversion instruction, ±MAX is stored in the floating point register.
   For the comparison instruction, the FCR:FCC:U flag is set.
   The FCR:ECF:V flag is set.

2. Division-by-Zero exception (Division by Zero)
   This exception occurs when performing division by zero. If this exception occurs, the following operations are carried out.
   
   [FCR:EEF:Z=1]
   Writing to the floating point register is prohibited.
   The FCR:CEF.Z flag is set to generate this exception.
   
   [FCR:EEF:Z=0]
   The infinity is stored in the floating point register. If the sign is the same, it is set to a positive sign. If the sign is different, it is set to a negative sign.
   The FCR:ECF:Z flag is set.
3. Overflow exception (Overflow)

This exception occurs when, during calculation, the exponent exceeds the maximum that can be expressed in the specified format. If this exception occurs, the following operations are carried out.

[FCR:EEF:O=1]
Writing to the floating point register is prohibited.
The FCR:CEF:O flag is set to generate this exception.

[FCR:EEF:O=0]
As shown in the following Table C.2-1, the value is written to the floating point register based on the rounding mode.
The FCR:ECF:O flag is set.

<table>
<thead>
<tr>
<th>Rounding mode (FCR:RM)</th>
<th>Output result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Positive overflow</td>
</tr>
<tr>
<td>00B (Latest value)</td>
<td>+∞</td>
</tr>
<tr>
<td>01B (Zero)</td>
<td>+MAX</td>
</tr>
<tr>
<td>10B (+∞)</td>
<td>+∞</td>
</tr>
<tr>
<td>11B (-∞)</td>
<td>+MAX</td>
</tr>
</tbody>
</table>

4. Underflow exception (Underflow)

This exception occurs when the rounding result is incorrect and the absolute value is less than the minimum normalized number in the specified format. In concrete terms, this exception occurs when the exponent is set to 0 while normalizing a significand or when the exponent is set to a negative value during multiplication or division. This exception also occurs when a specific value is set to the minimum normalized number after round processing while it is an unnormalized number before round processing (pre-rounding rule). This exception does not occur when the result of non-round processing is correct and set to zero. If this exception occurs, the following operations are carried out.

[FCR:EEF:U=1]
Writing to the floating point register is prohibited.
The FCR:CEF:U flag is set to generate this exception.

[FCR:EEF:U=0]
Zero is always stored in the floating point register. (Zero flush)
The FCR:ECF:U flag is set.

5. Inexact exception (Inexact)

This exception occurs when the rounding result is incorrect (including a case where an Underflow exception is detected at FCR:EEF:U = 0 when a unnormalized number is flushed to zero) or when overflow has occurred because an Overflow exception is invalid (including a case where overflow has occurred as a result of round processing). If this exception occurs, the following operations are carried out.

[FCR:EEF:X=1]
Writing to the floating point register is prohibited.
The FCR:CEF:X flag is set to generate this exception.
6. Unnormalized Number Input exception (Denormalized Number Input)
This exception occurs when an unnormalized number is specified in the input operand. If this exception occurs, the following operations are carried out.

[FCR:EEF:D=1]
Writing to the floating point register is prohibited.
The FCR:CEF:D flag is set to generate this exception.

[FCR:EEF:D=0]
The input operand that is set to an unnormalized number is flushed to zero before calculation.
The FCR:ECF:D flag is set.

C.3 Round Processing
Rounding processing conforms to IEEE754. A significand is expressed by 24 bits (single-precision). If a significand of calculation result is expressed by the number of bits greater than 24 bits (including unnormalized numbers), round processing is performed to obtain the approximate value of 24 bits (single-precision). The following explains round processing.

The calculation result (S) of the significand is defined as follows (see next if the guard bit is omitted (already processed)).

**Figure C.3-1 Calculation Result of Significand including Guard Bit**

<table>
<thead>
<tr>
<th>26</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>significand (p)</td>
<td>g</td>
<td>r</td>
<td>s</td>
<td></td>
</tr>
</tbody>
</table>

Here, "g" indicates a guard bit, "r" indicates a round bit, "s" indicates a sticky bit, and "p" indicates a significand.

Next, obtain the OR value (r ∪ s) between "r" and "s". Then set the result as "s" and set "g" as "r" again.

**Figure C.3-2 Calculation Result of Significand omitting Guard Bit**

<table>
<thead>
<tr>
<th>25</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>significand (p)</td>
<td>r</td>
<td>s</td>
<td></td>
</tr>
</tbody>
</table>

Perform round processing based on the following Table C.3-1. Here, "S" indicates the calculation result of the significand, "r" indicates a round bit, "s" indicates a sticky bit, and "LSB" indicates the LSB of "p". ("!s" indicates the reversal value of "s".)
## Table C.3-1  Rounding Mode and Rounding Processing of Significand

<table>
<thead>
<tr>
<th>Rounding mode (FCR.RM)</th>
<th>S ≥ 0</th>
<th>S &lt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00B (Latest value)</td>
<td>&quot;p+1&quot; if &quot;r^1s^LSB&quot; or &quot;r^s&quot; is true</td>
<td>&quot;p+1&quot; if &quot;r^1s^LSB&quot; or &quot;r^s&quot; is true</td>
</tr>
<tr>
<td>01B (Zero)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10B (+∞)</td>
<td>&quot;p+1&quot; if &quot;r∪s&quot; is true</td>
<td></td>
</tr>
<tr>
<td>11B (-∞)</td>
<td></td>
<td>&quot;p+1&quot; if &quot;r∪s&quot; is true</td>
</tr>
</tbody>
</table>

The blank column means that the "p" value is used as the result.
FR81 Family

INDEX

The index follows on the next page.
This is listed in alphabetic order.
Index

Numerics

20-bit Addressing

20-bit Addressing Area & 32-bit Addressing Area

...............................................................11

32-bit Addressing

20-bit Addressing Area & 32-bit Addressing Area

...............................................................11

A

Access

Data Access.........................................................14
Program Access ...................................................14

ADD

ADD (Add 4bit Immediate Data to Destination
Register)............................................................105
ADD (Add Word Data of Source Register to
Destination Register) ...........................................107
ADD2 (Add 4bit Immediate Data to Destination
Register)............................................................109

ADDC

ADDC (Add Word Data of Source Register and Carry
Bit to Destination Register) ..............................111

ADDN

ADDN (Add Immediate Data to Destination Register)
.................................................................113
ADDN (Add Word Data of Source Register to
Destination Register) ...........................................115
ADDN2 (Add Immediate Data to Destination
Register)............................................................117

Address Space

Address Space...........................................................8

Addressing

20-bit Addressing Area & 32-bit Addressing Area

.................................................................11

Addressing Formats

Addressing Formats.................................................86

ADDSP

ADDSP (Add Stack Pointer and Immediate Data)
.................................................................119

Alignment

Word Alignment ...................................................14

AND

AND (And Word Data of Source Register to Data
in Memory)..........................................................121
AND (And Word Data of Source Register to
Destination Register) ............................................123

ANDB

ANDB (And Byte Data of Source Register to Data
in Memory)..........................................................125

ANDCCR

ANDCCR (And Condition Code Register and
Immediate Data).................................................127

ANDH

ANDH (And Halfword Data of Source Register to
Data in Memory)..................................................129
FR81 Family

Arithmetic shift
ASR (Arithmetic shift to the Right Direction) ..................................................131, 133
ASR2 (Arithmetic shift to the Right Direction) ..................................................135
ASR
ASR (Arithmetic shift to the Right Direction) ..................................................131, 133
ASR2 (Arithmetic shift to the Right Direction) ..................................................135

B
BANDH
BANDH (And 4bit Immediate Data to Higher 4bit of Byte Data in Memory) ..........137
BANDL
BANDL (And 4bit Immediate Data to Lower 4bit of Byte Data in Memory) ..........139

Bcc
Bcc (Branch relative if Condition satisfied) ..................................................141
Bcc:D
Bcc:D (Branch relative if Condition satisfied) ..................................................143

BEORH
BEORH (Eor 4bit Immediate Data to Higher 4bit of Byte Data in Memory) ..........145
BEORL
BEORL (Eor 4bit Immediate Data to Lower 4bit of Byte Data in Memory) ..........147

BORH
BORH (Or 4bit Immediate Data to Higher 4bit of Byte Data in Memory) ..........149
BORL
BORL (Or 4bit Immediate Data to Lower 4bit of Byte Data in Memory) ..........151

Branch
Branching Instructions .................................................................97
Delayed Branching Instructions ..........................................................97
Delayed branching processing .........................................................78
Example of branching with non-delayed branching instructions .................78
Example of processing of delayed branching instruction ..........................79
Non-Delayed Branching Instructions ..................................................99
Specific example of Delayed Branching Instructions ..................................98

Branching Instructions
Branching Instructions and Delay Slot ..................................................97

BTSTH
BTSTH (Test Higher 4bit of Byte Data in Memory) .........................................153

BTSTL
BTSTL (Test Lower 4bit of Byte Data in Memory) .........................................155

Bypassing
Register Bypassing .................................................................74

Byte Data
ANDB (And Byte Data of Source Register to Data in Memory) ........................125
BTSTH (Test Higher 4bit of Byte Data in Memory) .........................................153
BTSTL (Test Lower 4bit of Byte Data in Memory) .........................................155

Byte Data
DMOV (Move Byte Data from Direct Address to Post Increment Register Indirect Address) ..........................................................199
DMOV (Move Byte Data from Direct Address to Register) ..........................195
DMOV (Move Byte Data from Post Increment Register Indirect Address to Direct Address) ..........................................................201
DMOV (Move Byte Data from Register to Direct Address) ..........................197
EORB (Exclusive Or Byte Data of Source Register to Data in Memory) ..........217
EXTLSB (Sign Extend from Byte Data to Word Data) ..................................221
EXTLB (Sign Extend from Byte Data to Word Data) ....................................223
EXTLUB (Unsign Extend from Byte Data to Word Data) ..............................225
EXTLUH (Unsign Extend from Byte Data to Word Data) ..............................227
LDUB (Load Byte Data in Memory to Register) ..........................................306, 308, 310
ORB (Or Byte Data of Source Register to Data in Memory) ..........................360
STB (Store Byte Data in Register to Memory) ..........................................394, 396, 398
XCHB (Exchange Byte Data) .................................................................420

Byte Order
Byte Order .................................................................13

C
CALL
CALL (Call Subroutine) .................................................................157, 159
CALL:D
CALL:D (Call Subroutine) .................................................................161, 163

Carry Bit
ADDC (Add Word Data of Source Register and Carry Bit to Destination Register) ..........................................................111
**FR81 Family**

**CCR**
- Condition Code Register (CCR) .................................. 21, 23

**CMP**
- CMP (Compare Immediate Data and Destination Register) .................................. 165  
- CMP (Compare Word Data in Source Register and Destination Register) .................. 167  
- CMP2 (Compare Immediate Data and Destination Register) .................................. 169

**Condition Code Register**
- ANDCCR (And Condition Code Register and Immediate Data) .......................... 127  
- Condition Code Register (CCR) .................................. 23  
- ORCCR (Or Condition Code Register and Immediate Data) ............................. 362

**Correction**
- DIV2 (Correction When Remain is 0) ................................ 177  
- DIV3 (Correction When Remain is 0) ................................ 179  
- DIV4S (Correction Answer for Signed Division) ........................................ 181

**CPU**
- Features of FR80 Family CPU ........................................ 2  
- FR80 Family CPU Register Configuration ................................ 16

**D**

**Data Access**
- Data Access ................................................................. 14

**Data Structure**
- Data Structure ............................................................ 12

**Dedicated Registers**
- Configuration of Dedicated Registers ................................ 19  
- Dedicated Registers ....................................................... 19

**Delay Slot**
- Branching Instructions and Delay Slot ................................... 97

**Delayed Branching**
- Delayed branching processing ........................................... 78

**Delayed Branching Instruction**
- Delayed Branching Instructions .......................................... 97  
- Example of processing of delayed branching instruction ......................... 79  
- Specific example of Delayed Branching Instructions .............................. 98

**Destination Register**
- ADD (Add 4bit Immediate Data to Destination Register) .......................... 105  
- ADD2 (Add 4bit Immediate Data to Destination Register) .......................... 109  
- ADDN (Add Immediate Data to Destination Register) .............................. 113  
- ADDN2 (Add Immediate Data to Destination Register) ............................ 117  
- CMP (Compare Immediate Data and Destination Register) .......................... 165

**DIV**
- DIV0S (Initial Setting Up for Signed Division) .................................. 171  
- DIV0U (Initial Setting Up for Unsigned Division) .................................. 173  
- DIV1 (Main Process of Division) .......................................... 175  
- DIV2 (Correction When Remain is 0) ...................................... 177  
- DIV3 (Correction When Remain is 0) ...................................... 179  
- DIV4S (Correction Answer for Signed Division) .................................. 181

**Division**
- DIV0S (Initial Setting Up for Signed Division) .................................. 171  
- DIV0U (Initial Setting Up for Unsigned Division) .................................. 173  
- DIV1 (Main Process of Division) .......................................... 175  
- DIV4S (Correction Answer for Signed Division) .................................. 181  
- Signed Division ............................................................ 100  
- Step Division Instructions ................................................ 100  
- Unsigned Division .......................................................... 101

**Direct Address**
- Direct Address Area ..................................................... 8  
- DMOV (Move Word Data from Direct Address to Post Increment Register Indirect Address) .................................................. 187  
- DMOV (Move Word Data from Direct Address to Pre Decrement Register Indirect Address) .............................................. 191  
- DMOV (Move Word Data from Direct Address to Register) .......................... 183  
- DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address) .................................................. 189  
- DMOV (Move Word Data from Register to Direct Address) .......................... 185  
- DMOVB (Move Byte Data from Direct Address to Post Increment Register Indirect Address) .................................................. 199  
- DMOVB (Move Byte Data from Direct Address to Register) .......................... 195  
- DMOVB (Move Byte Data from Register to Direct Address) .......................... 197  
- DMOVH (Move Halfword Data from Direct Address to Register) .................. 203  
- DMOVH (Move Halfword Data from Direct Address to Post Increment Register Indirect Address) .................................................. 207

**DMOV**
- ...
### FR81 Family

#### DMOV
- DMOV (Move Word Data from Direct Address to Post Increment Register Indirect Address) ..........................................................187
- DMOV (Move Word Data from Direct Address to Pre Decrement Register Indirect Address) ..........................................................191
- DMOV (Move Word Data from Direct Address to Register) .............................................183
- DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address) ..........................................................189, 193
- DMOV (Move Word Data from Register to Direct Address) .............................................185

#### DMOVB
- DMOVB (Move Byte Data from Direct Address to Post Increment Register Indirect Address) ..........................................................199
- DMOVB (Move Byte Data from Direct Address to Register) .............................................195
- DMOVB (Move Byte Data from Post Increment Register Indirect Address to Direct Address) ..........................................................201
- DMOVB (Move Byte Data from Register to Direct Address) .............................................197

#### DMOVH
- DMOVH (Move Halfword Data from Direct Address to Register) .............................................203
- DMOVH (Move Halfword Data from Direct Address to Post Increment Register Indirect Address) ..................................................207
- DMOVH (Move Halfword Data from Post Increment Register Indirect Address to Direct Address) ..................................................209
- DMOVH (Move Halfword Data from Register to Direct Address) .............................................205

#### E

#### EIT
- Basic Operations in EIT Processing.........................43
- EIT Processing Sequence .................................................44
- Multiple EIT Processing..............................................60
- Multiple EIT processing and Priority Levels.........60
- Priority Levels of EIT Requests .......................61
- Recovery from EIT Processing.........................45
- Types of EIT Processing and Prior Preparation .......43

#### Emulator
- INTE (Software Interrupt for Emulator) .................273

#### ENTER
- ENTER (Enter Function) .................................................211

#### EOR
- BEORH (Eor 4bit Immediate Data to Higher 4bit of Byte Data in Memory) ..................145
- EOR (Exclusive Or Word Data of Source Register to Destination Register) ..............215
- EOR (Exclusive Or Word Data of Source Register to Data in Memory) ..................213

#### EORB
- EORB (Exclusive Or Byte Data of Source Register to Data in Memory) ..............217

#### EORH
- EORH (Exclusive Or Halfword Data of Source Register to Data in Memory) .............219

#### Exception
- Exception Processing.................................................48

#### Exchange
- XCHB (Exchange Byte Data) .............................................420

#### Exclusive Or
- EOR (Exclusive Or Word Data of Source Register to Destination Register) ..............215
- EOR (Exclusive Or Word Data of Source Register to Data in Memory) ..................213
- EORH (Exclusive Or Halfword Data of Source Register to Data in Memory) .............219

#### Extend
- EXTSB (Sign Extend from Byte Data to Word Data) .............................................221
- EXTSH (Sign Extend from Byte Data to Word Data) .............................................223
- EXTUB (Unsign Extend from Byte Data to Word Data) .............................................225

#### F

#### FABSs
- FABSs (Single Precision Floating Point Absolute Value) .............................................229

#### FADDs
- FADDs (Single Precision Floating Point Add) .............................................230
### FBcc
- FBcc (Floating Point Conditional Branch) .................. 232
- FBcc:D (Floating Point Conditional Branch with Delay Slot) .................. 234

### FCMPs
- FCMPs (Single Precision Floating Point Compare) .................. 236

### FDIVs
- FDIVs (Single Precision Floating Point Division) .................. 238

### First One bit
- SRCH1 (Search First One bit position distance From MSB) .................. 375

### First Zero bit
- SRCH0 (Search First Zero bit position distance From MSB) .................. 373

### FTOs
- FTOs (Convert from Integer to Single Precision Floating Point) .................. 240

### FSQRTs
- FSQRTs (Single Precision Floating Point Square Root) .................. 258

### FST
- FST (Single Precision Floating Point Data Store) .................. 259, 260, 261, 262, 263
- FST (Store Word Data in Floating Point Register to Memory) .................. 264

### FSM
- FSM (Single Precision Floating Point Data Store from Multiple Register) .................. 265

### FSUBs
- FSUBs (Single Precision Floating Point Subtract) .................. 269

### G

#### General Interrupt
- General interrupts .................. 53

#### General-purpose Registers
- Configuration of General-purpose Registers .................. 17
- General-purpose Registers .................. 17
- Interlocking produced by reference to R15 and General-purpose Registers after Changing the Stack flag (S flag) .................. 75
- Special Usage of General-purpose Registers ........... 18

### H

#### Half Word Data
- Half Word Data .................. 12

#### Halfword Data
- ANDH (And Halfword Data of Source Register to Data in Memory) .................. 129
- DMOVH (Move Halfword Data from Direct Address to Register) .................. 203
- DMOVH (Move Halfword Data from Direct Address to Post Increment Register Indirect Address) .................. 207
- DMOVH (Move Halfword Data from Post Increment Register Indirect Address to Direct Address) .................. 209
- DMOVH (Move Halfword Data from Register to Direct Address) .................. 205
- EORH (Exclusive Or Halfword Data of Source Register to Data in Memory) .................. 219
- LDUH (Load Halfword Data in Memory to Register) .................. 313, 315, 317
- MULH (Multiply Halfword Data) .................. 348

### FR81 Family
- Features of FR81 Family CPU .................. 2
- FR81 Family CPU Register Configuration .................. 16

### FSQRTs
- FSQRTs (Single Precision Floating Point Square Root) .................. 258

### FST
- FST (Single Precision Floating Point Data Store) .................. 259, 260, 261, 262, 263
- FST (Store Word Data in Floating Point Register to Memory) .................. 264

### FSM
- FSM (Single Precision Floating Point Data Store from Multiple Register) .................. 265

### FSUBs
- FSUBs (Single Precision Floating Point Subtract) .................. 269
FR81 Family

MULUH (Multiply Unsigned Halfword Data) ........................................................................352
ORH (Or Halfword Data of Source Register to Data in Memory) ........................................364
STH (Store Halfword Data in Register to Memory) .................................................................401, 403, 405

hazard

Occurrence of register hazard ...........................................74
Register hazards ..............................................................74

I

Timing when the interrupt enable flag (I) is requested
..............................................................................63

ILM

Interrupt Level Mask Register (ILM) ...........................................22

Immediate 20bit Data

LDI:20 (Load Immediate 20bit Data to Destination Register) ...........................................296

Immediate 32 bit Data

LDI:32 (Load Immediate 32 bit Data to Destination Register) ...........................................298

Immediate 8bit Data

LDI:8 (Load Immediate 8bit Data to Destination Register) ..................................................300

Immediate Data

ADD (Add 4bit Immediate Data to Destination Register) .........................................................105
ADD2 (Add 4bit Immediate Data to Destination Register) .....................................................109
ADDN (Add Immediate Data to Destination Register) ..........................................................113
ADDN2 (Add Immediate Data to Destination Register) .........................................................117
ADDSP (Add Stack Pointer and Immediate Data) .................................................................119
BANDH (And 4bit Immediate Data to Higher 4bit of Byte Data in Memory) .....................137
BANDL (And 4bit Immediate Data to Lower 4bit of Byte Data in Memory) .....................139
BEORH (Eor 4bit Immediate Data to Higher 4bit of Byte Data in Memory) .....................145
BEORL (Eor 4bit Immediate Data to Lower 4bit of Byte Data in Memory) .....................147
BORH (Or 4bit Immediate Data to Higher 4bit of Byte Data in Memory) .........................149
BORL (Or 4bit Immediate Data to Lower 4bit of Byte Data in Memory) .........................151
CMP (Compare Immediate Data and Destination Register) ...............................................165
CMP2 (Compare Immediate Data and Destination Register) ..............................................169
ORCCR (Or Condition Code Register and Immediate Data) .............................................362
STILM (Set Immediate Data to Interrupt Level Mask Register) ........................................408

Increment Register

DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address)
..................................................................................189, 193
DMOVB (Move Byte Data from Post Increment Register Indirect Address to Direct Address)
..................................................................................201

Indirect Address

DMOV (Move Word Data from Direct Address to Post Increment Register Indirect Address)
............................................................................................187
DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address)
............................................................................................189

Instruction

“INT”Instructions ........................................................................57
Branching Instructions and Delay Slot .........................................97
Delayed Branching Instructions .................................................97
Example of branching with non-delayed branching instructions ..................................78
Example of processing of delayed branching instruction ......................79
Instruction execution based on Pipeline ....................................70
INTE Instruction ..............................................................57
Non-Delayed Branching Instructions ..........................................99
Read-Modify-Write type Instructions .........................................96
Specific example of Delayed Branching Instructions .........................98
Step Division Instructions .....................................................100

Instruction Format

Instruction Formats ...................................................................87
Instructions Formats ..............................................................85

Instruction System

Instruction System ....................................................................82

Instructions Notation Formats

Instructions Notation Formats ...................................................85

INT

“INT” Instructions .......................................................................57
INT (Software Interrupt) .........................................................271

INTE

INTE (Software Interrupt for Emulator) ........................................273
INTE Instruction ..............................................................57

Interlocking

Interlocking .............................................................................75
Interlocking produced by reference to R15 and General-purpose Registers after Changing
the Stack flag (S flag) .....75

Interrupt

General interrupts .....................................................................53
INT (Software Interrupt) .........................................................271
INTE (Software Interrupt for Emulator) ........................................273
Interrupts .................................................................53
Mismatch in Acceptance and Cancellation of Interrupt ............................................................73
Non-maskable Interrupts (NMI) ..................................................55
Pipeline Operation and Interrupt Processing ..................................................73
Points of Caution while using User Interrupts ..................................................66
Preparation while using user interrupts ..................................................65
Processing during an Interrupt Processing Routine ..................................................66
RETI (Return from Interrupt) ..................................................370
Usage Sequence of User Interrupts ..................................................65
interrupt enable flag
Timing when the interrupt enable flag (I) is requested ..................................................63
Interrupt Level Mask Register
Interrupt Level Mask Register (ILM) ..................................................22
Timing of Reflection of Interrupt Level Mask Register (ILM) ..................................................64
Interrupt Processing Routine
Processing during an Interrupt Processing Routine ..................................................66

J
JMP
JMP (Jump) ..................................................275
JMP:D
JMP:D (Jump) ..................................................277
Jump
JMP (Jump) ..................................................275
JMP:D (Jump) ..................................................277

L
LCALL
LCALL (Long Call Subroutine) ..................................................279
LCALL:D (Long Call Subroutine) ..................................................280
LD
LD (Load Word Data in Memory to Program Status Register) ..................................................294
LD (Load Word Data in Memory to Register) ..................................................281, 283, 285, 287, 289, 292
LDI:20
LDI:20 (Load Immediate 20bit Data to Destination Register) ..................................................296
LDI:32
LDI:32 (Load Immediate 32 bit Data to Destination Register) ..................................................298
LDI:8
LDI:8 (Load Immediate 8bit Data to Destination Register) ..................................................300
LDM0
LDM0 (Load Multiple Registers) ..................................................302
LDM1
LDM1 (Load Multiple Registers) ..................................................304
LDU
LDU (Load Byte Data in Memory to Register) ..................................................306, 308, 310
LDUH
LDUH (Load Halfword Data in Memory to Register) ..................................................313, 315, 317
LEAVE
LEAVE (Leave Function) ..................................................320
Load
LD
LD (Load Word Data in Memory to Program Status Register) ..................................................294
LD (Load Word Data in Memory to Register) ..................................................281, 283, 285, 287, 289, 292
LDI:20
LDI:20 (Load Immediate 20bit Data to Destination Register) ..................................................296
LDI:32
LDI:32 (Load Immediate 32 bit Data to Destination Register) ..................................................298
LDI:8
LDI:8 (Load Immediate 8bit Data to Destination Register) ..................................................300
LDM0
LDM0 (Load Multiple Registers) ..................................................302
LDM1
LDM1 (Load Multiple Registers) ..................................................304
LDU
LDU (Load Byte Data in Memory to Register) ..................................................306, 308, 310
LDUH
LDUH (Load Halfword Data in Memory to Register) ..................................................313, 315, 317
Logical Shift
LSL
LSL (Logical Shift to the Left Direction) ..................................................322, 324
LSL2
LSL2 (Logical Shift to the Left Direction) ..................................................326
LSR
LSR (Logical Shift to the Right Direction) ..................................................328, 330
LSR2
LSR2 (Logical Shift to the Right Direction) ..................................................332

M
MDH
MDH (Multiplication/Division Register (MDH, MDL)) ..................................................30
MDL
MDL (Multiplication/Division Register (MDH, MDL)) ..................................................30
MOV
MOV (Move Word Data in Floating Point Register to General Purpose Register) ..................................................345
MOV (Move Word Data in General Purpose Register to Floating Point Register) ..................................................344
## FR81 Family

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>Move Word Data in Program Status Register to Destination Register</td>
<td>338</td>
</tr>
<tr>
<td>MOV</td>
<td>Move Word Data in Source Register to Destination Register</td>
<td>334, 336, 340</td>
</tr>
<tr>
<td>MOV</td>
<td>Move Word Data in Source Register to Program Status Register</td>
<td>342</td>
</tr>
<tr>
<td>MOV</td>
<td>Move Word Data in Program Status Register to Destination Register</td>
<td>338</td>
</tr>
<tr>
<td>MOV</td>
<td>Move Word Data in Source Register to Destination Register</td>
<td>334, 336, 340</td>
</tr>
<tr>
<td>MOV</td>
<td>Move Word Data in Source Register to Program Status Register</td>
<td>342</td>
</tr>
<tr>
<td>MSB</td>
<td>SRCH0 (Search First Zero bit position distance From MSB)</td>
<td>373</td>
</tr>
<tr>
<td></td>
<td>SRCH1 (Search First One bit position distance From MSB)</td>
<td>375</td>
</tr>
<tr>
<td>MUL</td>
<td>MUL (Multiply Word Data)</td>
<td>346</td>
</tr>
<tr>
<td>MULH</td>
<td>MULH (Multiply Halfword Data)</td>
<td>348</td>
</tr>
<tr>
<td>Multiply</td>
<td>Multiple EIT Processing</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>Multiple EIT processing and Priority Levels</td>
<td>60</td>
</tr>
<tr>
<td>Multiple Registers</td>
<td>LDM0 (Load Multiple Registers)</td>
<td>302</td>
</tr>
<tr>
<td></td>
<td>LDM1 (Load Multiple Registers)</td>
<td>304</td>
</tr>
<tr>
<td></td>
<td>STM0 (Store Multiple Registers)</td>
<td>410</td>
</tr>
<tr>
<td></td>
<td>STM1 (Store Multiple Registers)</td>
<td>412</td>
</tr>
<tr>
<td>Multiplication/Division Register</td>
<td>Multiplication/Division Register (MDH, MDL)</td>
<td>30</td>
</tr>
<tr>
<td>Multiply</td>
<td>MUL (Multiply Word Data)</td>
<td>346</td>
</tr>
<tr>
<td></td>
<td>MULH (Multiply Halfword Data)</td>
<td>348</td>
</tr>
<tr>
<td></td>
<td>MULU (Multiply Unsigned Word Data)</td>
<td>350</td>
</tr>
<tr>
<td></td>
<td>MULUH (Multiply Unsigned Halfword Data)</td>
<td>352</td>
</tr>
<tr>
<td>MULU</td>
<td>MULU (Multiply Unsigned Word Data)</td>
<td>350</td>
</tr>
<tr>
<td>MULUH</td>
<td>MULUH (Multiply Unsigned Halfword Data)</td>
<td>352</td>
</tr>
<tr>
<td>N</td>
<td>NMI</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>Non-maskable Interrupts (NMI)</td>
<td>55</td>
</tr>
<tr>
<td>No Operation</td>
<td>NOP (No Operation)</td>
<td>354</td>
</tr>
<tr>
<td></td>
<td>Non-block loading</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>Non-delayed branching</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>Example of branching with non-delayed branching instructions</td>
<td>78</td>
</tr>
<tr>
<td>Non-Delayed Branching Instructions</td>
<td>Non-Delayed Branching Instructions</td>
<td>99</td>
</tr>
<tr>
<td>Non-maskable Interrupts</td>
<td>Non-maskable Interrupts (NMI)</td>
<td>55</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP (No Operation)</td>
<td>354</td>
</tr>
<tr>
<td>O</td>
<td>OR</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td>BORH (Or 4bit Immediate Data to Higher 4bit of Byte Data in Memory)</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td>BORL (Or 4bit Immediate Data to Lower 4bit of Byte Data in Memory)</td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>OR (Or Word Data of Source Register to Data in Memory)</td>
<td>356</td>
</tr>
<tr>
<td></td>
<td>OR (Or Word Data of Source Register to Destination Register)</td>
<td>358</td>
</tr>
<tr>
<td>ORB</td>
<td>ORB (Or Byte Data of Source Register to Data in Memory)</td>
<td>360</td>
</tr>
<tr>
<td>ORCCR</td>
<td>ORCCR (Or Condition Code Register and Immediate Data)</td>
<td>362</td>
</tr>
<tr>
<td>ORH</td>
<td>ORH (Or Halfword Data of Source Register to Data in Memory)</td>
<td>364</td>
</tr>
<tr>
<td>P</td>
<td>PC</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Program Counter (PC)</td>
<td>20</td>
</tr>
<tr>
<td>Pipeline</td>
<td>How to prevent mismatched pipeline conditions?</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>Instruction execution based on Pipeline</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>Pipeline Operation and Interrupt Processing</td>
<td>73</td>
</tr>
<tr>
<td>Pointer</td>
<td>ADDSP (Add Stack Pointer and Immediate Data)</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>ADDSP (Add Stack Pointer and Immediate Data)</td>
<td>119</td>
</tr>
<tr>
<td>Post</td>
<td>DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address)</td>
<td>193</td>
</tr>
<tr>
<td></td>
<td>DMOVB (Move Byte Data from Direct Address to Post Increment Register Indirect Address)</td>
<td>199</td>
</tr>
<tr>
<td></td>
<td>DMOVB (Move Byte Data from Post Increment Register Indirect Address to Direct Address)</td>
<td>201</td>
</tr>
</tbody>
</table>
DMOVH (Move Halfword Data from Direct Address to Post Increment Register Indirect Address) .......................................................... 207
DMOVH (Move Halfword Data from Post Increment Register Indirect Address to Direct Address) .......................................................... 209

Prior Preparation
Types of EIT Processing and Prior Preparation ........................................................................... 43
Priority Levels
Multiple EIT processing and Priority Levels .................................................................................. 60
Priority Levels of EIT Requests .................................................................................................... 61

Processing
Multiple EIT Processing ..................................................................................................................... 60
Processing
Multiple EIT processing and Priority Levels .................................................................................. 60

Program Access
Program Access .................................................................................................................................. 14
Program Counter
Program Counter (PC) ...................................................................................................................... 20
Program Status
LD (Load Word Data in Memory to Program Status Register) ......................................................... 294
MOV (Move Word Data in Program Status Register to Destination Register) ................................. 338
Program Status (PS) ....................................................................................................................... 20

Program Status Register
ST (Store Word Data in Program Status Register to Memory) ...................................................... 392
PS
Program Status (PS) ....................................................................................................................... 20

S
S flag
Interlocking produced by reference to R15 and General-purpose Registers after Changing the Stack flag (S flag) .................................................................................. 75

Search
SRCH0 (Search First Zero bit position distance From MSB) .......................................................... 373
SRCH1 (Search First One bit position distance From MSB) .......................................................... 375
SRCHC (Search First bit value change position distance From MSB) ........................................ 377

Sign Extend
EXTSB (Sign Extend from Byte Data to Word Data) ....................................................................... 221
EXTSH (Sign Extend from Byte Data to Word Data) ..................................................................... 223

Signed Division
DIV0S (Initial Setting Up for Signed Division) ................................................................................ 171
DIV4S (Correction Answer for Signed Division) ............................................................................ 181
Signed Division ............................................................................................................................. 100
FR81 Family

Slot
Branching Instructions and Delay Slot .......... 97

Software Interrupt
INT (Software Interrupt) ......................... 271
INTE (Software Interrupt for Emulator) .......... 273

Source Register
ADD (Add Word Data of Source Register to Destination Register) ........ 107
ADDC (Add Word Data of Source Register and Carry Bit to Destination Register) ...... 111
ADDN (Add Word Data of Source Register to Destination Register) .................. 115
AND (And Word Data of Source Register to Data in Memory) ....................... 121
AND (And Word Data of Source Register to Destination Register) .................. 123
ANDB (And Byte Data of Source Register to Data in Memory) ....................... 125
ANDH (And Halfword Data of Source Register to Data in Memory) ................. 129
CMP (Compare Word Data in Source Register and Destination Register) ............ 167
EOR (Exclusive Or Word Data of Source Register to Destination Register) .......... 215
EOR (Exclusive Or Word Data of Source Register to Data in Memory) ................. 213
EORB (Exclusive Or Byte Data of Source Register to Data in Memory) ................. 217
EORH (Exclusive Or Halfword Data of Source Register to Data in Memory) ........ 219
MOV (Move Word Data in Source Register to Destination Register) .................. 334, 336, 340
MOV (Move Word Data in Source Register to Program Status Register) .......... 342
OR (Or Word Data of Source Register to Data in Memory) ........................... 356
OR (Or Word Data of Source Register to Destination Register) ...................... 358
ORB (Or Byte Data of Source Register to Data in Memory) ........................... 360
ORH (Or Halfword Data of Source Register to Data in Memory) ....................... 364
SUB (Subtract Word Data in Source Register from Destination Register) .......... 414
SUBC (Subtract Word Data in Source Register and Carry bit from Destination Register) ...... 416
SUBN (Subtract Word Data in Source Register from Destination Register) .......... 418

Special Usage
Special Usage of General-purpose Registers ........ 18
SRCH
SRCH0 (Search First Zero bit position distance From MSB) .......................... 373
SRCH1 (Search First One bit position distance From MSB) .......................... 375
SRCHC
SRCHC (Search First bit value change position distance From MSB) ......... 377

SSP
System Stack Pointer (SSP) ....................... 27
ST
ST (Store Word Data in Program Status Register to Memory) ..................... 392
ST (Store Word Data in Register to Memory) ........ 379, 381, 383, 385, 387, 389, 390

Stack flag
Interlocking produced by reference to R15 and General-purpose Registers after Changing the Stack flag (S flag) ............ 75

Stack Pointer
ADDSP (Add Stack Pointer and Immediate Data) ............................. 374, 394, 396, 398, 400
Relation between Stack Pointer and R15 ..................... 119

STB
STB (Store Byte Data in Register to Memory) .................. 394, 396, 398, 400

Step Division Instructions
Step Division Instructions ......................... 100

Step Trace
Step Trace Traps .................................. 58

STH
STH (Store Halfword Data in Register to Memory) .................. 401, 403, 405, 407

STILM
STILM (Set Immediate Data to Interrupt Level Mask Register) ........................ 408

STM
STM0 (Store Multiple Registers) ................. 410
STM1 (Store Multiple Registers) ................. 412

Store
ST (Store Word Data in Program Status Register to Memory) ..................... 392
ST (Store Word Data in Register to Memory) ........ 379, 381, 383, 385, 387, 390
STB (Store Byte Data in Register to Memory) .................. 394, 396, 398
STH (Store Halfword Data in Register to Memory) .................. 401, 403, 405
STM0 (Store Multiple Registers) ................. 410
STM1 (Store Multiple Registers) ................. 412

SUB
SUB (Subtract Word Data in Source Register from Destination Register) .......... 414
Subtract

SUBC (Subtract Word Data in Source Register and Carry bit from Destination Register) .................. 416

UBN

SUBN (Subtract Word Data in Source Register from Destination Register) ................... 418

Subroutine

CALL (Call Subroutine) ..................... 157, 159
CALLD (Call Subroutine) .................. 161, 163
RET (Return from Subroutine) .................. 366
RET:D (Return from Subroutine) .................. 368

System Condition Code Register

System Condition Code Register (SCR) ................. 25

System Stack Pointer

System Stack Pointer (SSP) ........................ 27

System Status Register

System Status Register (SSR) .................... 21

T

Table Base Register

Table Base Register (TBR) ....................... 29

Test

BTSTH (Test Higher 4bit of Byte Data in Memory) ................. 153
BTSTL (Test Lower 4bit of Byte Data in Memory) .................. 155

Trace

Step Trace Traps .......................... 58

Traps

Step Trace Traps .......................... 58
Traps ........................................... 57

TBR

Table Base Register (TBR) ....................... 29

U

Unsign Extend

EXTUB (Unsign Extend from Byte Data to Word Data) .................. 225
EXTUH (Unsign Extend from Byte Data to Word Data) .................. 227

Unsigned Division

DIV0U (Initial Setting Up for Unsigned Division) ............... 173

Unsigned Division

MULUH (Multiply Unsigned Halfword Data) .................. 352

Unsigned Word Data

MULU (Multiply Unsigned Word Data) .................. 350

User Interrupt

Points of Caution while using User Interrupts ....... 66
Preparation while using user interrupts ............... 65
Usage Sequence of User Interrupts .................. 65

User Stack Pointer

User Stack Pointer (USP) ........................ 28

USP

User Stack Pointer (USP) ........................ 28

V

value change

SRCHC (Search First bit value change position distance From MSB) .................. 377

Vector Table

Vector Table Area ........................... 9

W

Word Alignment

Word Alignment ....................... 14

Word Data

ADD (Add Word Data of Source Register to Destination Register) .................. 107
ADDC (Add Word Data of Source Register and Carry Bit to Destination Register) ........... 111
ADDN (Add Word Data of Source Register to Destination Register) .................. 115
AND (And Word Data of Source Register to Data in Memory) .................. 121
AND (And Word Data of Source Register to Destination Register) .................. 123
CMP (Compare Word Data in Source Register and Destination Register) .................. 167
DMOV (Move Word Data from Direct Address to Post Increment Register Indirect Address) .................. 187
DMOV (Move Word Data from Direct Address to Pre Decrement Register Indirect Address) .................. 191
DMOV (Move Word Data from Direct Address to Register) .................. 183
DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address) .................. 189, 193
DMOV (Move Word Data from Register to Direct Address) .................. 185
EOR (Exclusive Or Word Data of Source Register to Destination Register) ............... 215
FR81 Family

EOR (Exclusive Or Word Data of Source Register to Data in Memory) .......................213
EXTSB (Sign Extend from Byte Data to Word Data) ............................................221
EXTSH (Sign Extend from Byte Data to Word Data) ..........................................223
EXTUB (Unsign Extend from Byte Data to Word Data) ........................................225
EXTUH (Unsign Extend from Byte Data to Word Data) .......................................227
LD (Load Word Data in Memory to Program Status Register) ..........................294
LD (Load Word Data in Memory to Register) .............................................281, 283, 285, 287, 289, 292
MOV (Move Word Data in Program Status Register to Destination Register) ....338
MOV (Move Word Data in Source Register to Destination Register) ........334, 336, 340
MOV (Move Word Data in Source Register to Program Status Register) ....342
MUL (Multiply Word Data) .............................................................................346
MULU (Multiply Unsigned Word Data) .........................................................350
OR (Or Word Data of Source Register to Data in Memory) ..............................356
OR (Or Word Data of Source Register to Destination Register) ......................358
ST (Store Word Data in Program Status Register to Memory) .......................392
ST (Store Word Data in Register to Memory) ..................................................379, 381, 383, 385, 387, 390
SUB (Subtract Word Data in Source Register from Destination Register) ......414
SUBC (Subtract Word Data in Source Register and Carry bit from Destination Register) ..........................................................416
SUBN (Subtract Word Data in Source Register from Destination Register) ......418
X
XCHB
XCHB (Exchange Byte Data) ...........................................................................420
FR81 Family