To our customers,

Old Company Name in Catalogs and Other Documents

On April 1\textsuperscript{st}, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: [http://www.renesas.com](http://www.renesas.com)

April 1\textsuperscript{st}, 2010  
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation ([http://www.renesas.com](http://www.renesas.com))

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.

2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.

3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.

4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.

5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.

6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.

7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depend on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

“Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

“High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-terrorism systems; safety equipment; and medical equipment not specifically designed for life support.

“Specific”: Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g., artificial life support devices or systems), surgical implants, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.

8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.

9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.

10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.

11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.

12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.
To all our customers

Regarding the change of names mentioned in the document, such as Hitachi Electric and Hitachi XX, to Renesas Technology Corp.

The semiconductor operations of Mitsubishi Electric and Hitachi were transferred to Renesas Technology Corporation on April 1st 2003. These operations include microcomputer, logic, analog and discrete devices, and memory chips other than DRAMs (flash memory, SRAMs etc.) Accordingly, although Hitachi, Hitachi, Ltd., Hitachi Semiconductors, and other Hitachi brand names are mentioned in the document, these names have in fact all been changed to Renesas Technology Corp. Thank you for your understanding. Except for our corporate trademark, logo and corporate statement, no changes whatsoever have been made to the contents of the document, and these changes do not constitute any alteration to the contents of the document itself.

Renesas Technology Home Page: http://www.renesas.com

Renesas Technology Corp.
Customer Support Dept.
April 1, 2003
### Cautions

Keep safety first in your circuit designs!

1. Renesas Technology Corporation puts the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage. Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

### Notes regarding these materials

1. These materials are intended as a reference to assist our customers in the selection of the Renesas Technology Corporation product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation or a third party.

2. Renesas Technology Corporation assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.

3. All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor for the latest product information before purchasing a product listed herein.

The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corporation assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.

Please also pay attention to information published by Renesas Technology Corporation by various means, including the Renesas Technology Corporation Semiconductor home page (http://www.renesas.com).

4. When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.

5. Renesas Technology Corporation semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.

6. The prior written approval of Renesas Technology Corporation is necessary to reprint or reproduce in whole or in part these materials.

7. If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination. Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.

8. Please contact Renesas Technology Corporation for further details on these materials or the products contained therein.
Notice

When using this document, keep the following in mind:

1. This document may, wholly or partially, be subject to change without notice.

2. All rights are reserved: No one is permitted to reproduce or duplicate, in any form, the whole or part of this document without Hitachi’s permission.

3. Hitachi will not be held responsible for any damage to the user that may result from accidents or any other reasons during operation of the user’s unit according to this document.

4. Circuitry and other examples described herein are meant merely to indicate the characteristics and performance of Hitachi’s semiconductor products. Hitachi assumes no responsibility for any intellectual property claims or other problems that may result from applications based on the examples described herein.

5. No license is granted by implication or otherwise under any patents or other rights of any third party or Hitachi, Ltd.

6. MEDICAL APPLICATIONS: Hitachi’s products are not authorized for use in MEDICAL APPLICATIONS without the written consent of the appropriate officer of Hitachi’s sales company. Such use includes, but is not limited to, use in life support systems. Buyers of Hitachi’s products are requested to notify the relevant Hitachi sales offices when planning to use the products in MEDICAL APPLICATIONS.
Section 1 Move Data ........................................................................................................ 1
   1.1 Set Constants ........................................................................................................... 1
      1.1.1 Function....................................................................................................... 1
      1.1.2 Arguments ................................................................................................. 1
      1.1.3 Internal Register and Flag Changes....................................................... 1
      1.1.4 Specifications ......................................................................................... 2
      1.1.5 Note ........................................................................................................ 2
      1.1.6 Description ............................................................................................ 2
      1.1.7 Flowchart ............................................................................................. 5
      1.1.8 Program List ......................................................................................... 6
   1.2 Move Block 1 ........................................................................................................... 7
      1.2.1 Function................................................................................................... 7
      1.2.2 Arguments ............................................................................................. 7
      1.2.3 Internal Register and Flag Changes................................................... 7
      1.2.4 Specifications ..................................................................................... 8
      1.2.5 Note .................................................................................................... 8
      1.2.6 Description ......................................................................................... 8
      1.2.7 Flowchart .......................................................................................... 12
      1.2.8 Program List .................................................................................... 13
   1.3 Move Block 2 (Example of the EEPMOV Instruction) ........................................... 14
      1.3.1 Function.................................................................................................. 14
      1.3.2 Arguments ............................................................................................ 14
      1.3.3 Internal Register and Flag Changes................................................ 14
      1.3.4 Specifications .................................................................................... 15
      1.3.5 Note ................................................................................................... 15
      1.3.6 Description ........................................................................................ 15
      1.3.7 Flowchart .......................................................................................... 20
      1.3.8 Program List .................................................................................... 22
   1.4 Move Character Strings ......................................................................................... 23
      1.4.1 Function.................................................................................................. 23
      1.4.2 Arguments ............................................................................................ 23
      1.4.3 Internal Register and Flag Changes................................................ 23
      1.4.4 Specifications ................................................................................... 24
      1.4.5 Note ................................................................................................... 24
      1.4.6 Description ........................................................................................ 25
      1.4.7 Flowchart .......................................................................................... 28
      1.4.8 Program List .................................................................................... 29
7.1.7 Program List...........................................................................................................89
7.2 Subtraction of 32-Bit Binary Numbers...........................................................................90
  7.2.1 Function................................................................................................................90
  7.2.2 Arguments............................................................................................................90
  7.2.3 Internal Register and Flag Changes......................................................................90
  7.2.4 Specifications......................................................................................................91
  7.2.5 Description..........................................................................................................91
  7.2.6 Flowchart...........................................................................................................95
  7.2.7 Program List......................................................................................................96
7.3 Multiplication of 16-Bit Binary Numbers......................................................................97
  7.3.1 Function................................................................................................................97
  7.3.2 Arguments............................................................................................................97
  7.3.3 Internal Register and Flag Changes......................................................................97
  7.3.4 Specifications......................................................................................................98
  7.3.5 Description..........................................................................................................98
  7.3.6 Flowchart...........................................................................................................102
  7.3.7 Program List......................................................................................................103
7.4 Division of 32-Bit Binary Numbers...............................................................................104
  7.4.1 Function................................................................................................................104
  7.4.2 Arguments............................................................................................................104
  7.4.3 Internal Register and Flag Changes......................................................................104
  7.4.4 Specifications......................................................................................................105
  7.4.5 Description..........................................................................................................105
  7.4.6 Flowchart...........................................................................................................110
  7.4.7 Program List......................................................................................................112
7.5 Addition of Multiple-Precision Binary Numbers..........................................................113
  7.5.1 Function................................................................................................................113
  7.5.2 Arguments............................................................................................................113
  7.5.3 Internal Register and Flag Changes......................................................................113
  7.5.4 Specifications......................................................................................................114
  7.5.5 Notes....................................................................................................................114
  7.5.6 Description..........................................................................................................114
  7.5.7 Flowchart...........................................................................................................118
  7.5.8 Program List......................................................................................................120
7.6 Subtraction of Multiple-Precision Binary Numbers......................................................121
  7.6.1 Function................................................................................................................121
  7.6.2 Arguments............................................................................................................121
  7.6.3 Internal Register and Flag Changes......................................................................121
  7.6.4 Specifications......................................................................................................122
  7.6.5 Notes....................................................................................................................122
  7.6.6 Description..........................................................................................................122
  7.6.7 Flowchart...........................................................................................................126
  7.6.8 Program List......................................................................................................128

RENESSAS
7.7 Addition of 8-Digit BCD Numbers ................................................................. 129
  7.7.1 Function ........................................................................................................ 129
  7.7.2 Arguments ...................................................................................................... 129
  7.7.3 Internal Register and Flag Changes ............................................................... 129
  7.7.4 Specifications ............................................................................................... 130
  7.7.5 Description .................................................................................................... 130
  7.7.6 Flowchart .................................................................................................... 135
  7.7.7 Program List ............................................................................................... 136
7.8 Subtraction of 8-Digit BCD Numbers ............................................................... 137
  7.8.1 Function ........................................................................................................ 137
  7.8.2 Arguments ...................................................................................................... 137
  7.8.3 Internal Register and Flag Changes ............................................................... 137
  7.8.4 Specifications ............................................................................................... 138
  7.8.5 Description .................................................................................................... 138
  7.8.6 Flowchart .................................................................................................... 143
  7.8.7 Program List ............................................................................................... 144
7.9 Multiplication of 4-Digit BCD Numbers ........................................................... 145
  7.9.1 Function ........................................................................................................ 145
  7.9.2 Arguments ...................................................................................................... 145
  7.9.3 Internal Register and Flag Changes ............................................................... 145
  7.9.4 Specifications ............................................................................................... 146
  7.9.5 Notes ............................................................................................................. 146
  7.9.6 Description .................................................................................................... 146
  7.9.7 Flowchart .................................................................................................... 151
  7.9.8 Program List ............................................................................................... 153
7.10 Division of 8-Digit BCD Numbers .................................................................... 154
  7.10.1 Function ...................................................................................................... 154
  7.10.2 Arguments .................................................................................................... 154
  7.10.3 Internal Register and Flag Changes ............................................................. 154
  7.10.4 Specifications ............................................................................................. 155
  7.10.5 Notes .......................................................................................................... 155
  7.10.6 Description .................................................................................................. 156
  7.10.7 Flowchart ................................................................................................... 160
  7.10.8 Program List .............................................................................................. 163
7.11 Addition of Multiple-Precision BCD Numbers ................................................. 165
  7.11.1 Function ...................................................................................................... 165
  7.11.2 Arguments .................................................................................................. 165
  7.11.3 Internal Register and Flag Changes ............................................................. 165
  7.11.4 Specifications ............................................................................................. 166
  7.11.5 Notes .......................................................................................................... 166
  7.11.6 Description .................................................................................................. 166
  7.11.7 Flowchart ................................................................................................... 171
  7.11.8 Program List .............................................................................................. 173
7.12 Subtraction of Multiple-Precision BCD Numbers ............................................................ 174
  7.12.1 Function................................................................................................................ 174
  7.12.2 Arguments .......................................................................................................... 174
  7.12.3 Internal Register and Flag Changes...................................................................... 174
  7.12.4 Specifications .................................................................................................... 175
  7.12.5 Notes................................................................................................................... 175
  7.12.6 Description ....................................................................................................... 175
  7.12.7 Flowchart......................................................................................................... 179
  7.12.8 Program List..................................................................................................... 181

7.13 Addition of Signed 32-Bit Binary Numbers...................................................................... 182
  7.13.1 Function................................................................................................................ 182
  7.13.2 Arguments .......................................................................................................... 182
  7.13.3 Internal Register and Flag Changes...................................................................... 182
  7.13.4 Specifications .................................................................................................... 183
  7.13.5 Description ....................................................................................................... 183
  7.13.6 Flowchart......................................................................................................... 187
  7.13.7 Program List..................................................................................................... 189

7.14 Multiplication of Signed 16-Bit Binary Numbers ............................................................ 190
  7.14.1 Function................................................................................................................ 190
  7.14.2 Arguments .......................................................................................................... 190
  7.14.3 Internal Register and Flag Changes...................................................................... 190
  7.14.4 Specifications .................................................................................................... 191
  7.14.5 Notes................................................................................................................... 191
  7.14.6 Description ....................................................................................................... 191
  7.14.7 Flowchart......................................................................................................... 195
  7.14.8 Program List..................................................................................................... 197

7.15 Change of a Short Floating-Point Number to a Signed 32-Bit Binary Number................ 200
  7.15.1 Function................................................................................................................ 200
  7.15.2 Arguments .......................................................................................................... 200
  7.15.3 Internal Register and Flag Changes...................................................................... 200
  7.15.4 Specifications .................................................................................................... 201
  7.15.5 Notes................................................................................................................... 201
  7.15.6 Description ....................................................................................................... 201
  7.15.7 Flowchart......................................................................................................... 204
  7.15.8 Program List..................................................................................................... 207

7.16 Change of a Signed 32-Bit Binary Number to a Short Floating-Point Number................ 209
  7.16.1 Function................................................................................................................ 209
  7.16.2 Arguments .......................................................................................................... 209
  7.16.3 Internal Register and Flag Changes...................................................................... 209
  7.16.4 Specifications .................................................................................................... 210
  7.16.5 Notes................................................................................................................... 210
  7.16.6 Description ....................................................................................................... 210
  7.16.7 Flowchart......................................................................................................... 213
7.16.8  Program List ........................................................................................................ 216

7.17  Addition of Short Floating-Point Numbers ........................................................................ 218
  7.17.1  Function................................................................................................................ 218
  7.17.2  Arguments ............................................................................................................ 218
  7.17.3  Internal Register and Flag Changes...................................................................... 218
  7.17.4  Specifications ...................................................................................................... 219
  7.17.5  Notes..................................................................................................................... 219
  7.17.6  Description .......................................................................................................... 220
  7.17.7  Flowchart.............................................................................................................. 224
  7.17.8  Program List ........................................................................................................ 232

7.18  Multiplication of Short Floating-Point Numbers ...................................................................... 235
  7.18.1  Function................................................................................................................ 235
  7.18.2  Arguments ............................................................................................................ 235
  7.18.3  Internal Register and Flag Changes...................................................................... 235
  7.18.4  Specifications ...................................................................................................... 236
  7.18.5  Notes..................................................................................................................... 236
  7.18.6  Description .......................................................................................................... 236
  7.18.7  Flowchart.............................................................................................................. 241
  7.18.8  Program List ........................................................................................................ 251

7.19  Square Root of a 32-Bit Binary Number .............................................................................. 255
  7.19.1  Function................................................................................................................ 255
  7.19.2  Arguments ............................................................................................................ 255
  7.19.3  Internal Register and Flag Changes...................................................................... 255
  7.19.4  Specifications ...................................................................................................... 256
  7.19.5  Notes..................................................................................................................... 256
  7.19.6  Description .......................................................................................................... 256
  7.19.7  Flowchart.............................................................................................................. 260
  7.19.8  Program List ........................................................................................................ 263

Section 8  DECIMAL ↔ HEXADECIMAL CHANGE ........................................ 265

8.1  Change a 2-Byte Hexadecimal Number to a 5-Character BCD Number .............................. 265
  8.1.1  Function................................................................................................................ 265
  8.1.2  Arguments ............................................................................................................ 265
  8.1.3  Internal Register and Flag Changes...................................................................... 265
  8.1.4  Specifications ...................................................................................................... 266
  8.1.5  Description .......................................................................................................... 266
  8.1.6  Flowchart.............................................................................................................. 270
  8.1.7  Program List ........................................................................................................ 271

8.2  Change a 5-Character BCD Number to a 2-Byte Hexadecimal Number .............................. 272
  8.2.1  Function................................................................................................................ 272
  8.2.2  Arguments ............................................................................................................ 272
  8.2.3  Internal Register and Flag Changes...................................................................... 272
  8.2.4  Specifications ...................................................................................................... 273
<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2.5</td>
<td>Description</td>
<td>273</td>
</tr>
<tr>
<td>8.2.6</td>
<td>Flowchart</td>
<td>277</td>
</tr>
<tr>
<td>8.2.7</td>
<td>Program List</td>
<td>279</td>
</tr>
<tr>
<td>Section 9</td>
<td>Sorting</td>
<td>281</td>
</tr>
<tr>
<td>9.1</td>
<td>Set Constants</td>
<td>281</td>
</tr>
<tr>
<td>9.1.1</td>
<td>Function</td>
<td>281</td>
</tr>
<tr>
<td>9.1.2</td>
<td>Arguments</td>
<td>281</td>
</tr>
<tr>
<td>9.1.3</td>
<td>Internal Register and Flag Changes</td>
<td>281</td>
</tr>
<tr>
<td>9.1.4</td>
<td>Specifications</td>
<td>282</td>
</tr>
<tr>
<td>9.1.5</td>
<td>Note</td>
<td>282</td>
</tr>
<tr>
<td>9.1.6</td>
<td>Description</td>
<td>282</td>
</tr>
<tr>
<td>9.1.7</td>
<td>Flowchart</td>
<td>286</td>
</tr>
<tr>
<td>9.1.8</td>
<td>Program List</td>
<td>287</td>
</tr>
<tr>
<td>Section 10</td>
<td>ARRAY</td>
<td>289</td>
</tr>
<tr>
<td>10.1</td>
<td>2-Dimensional Array (ARRAY)</td>
<td>289</td>
</tr>
<tr>
<td>10.1.1</td>
<td>Function</td>
<td>289</td>
</tr>
<tr>
<td>10.1.2</td>
<td>Arguments</td>
<td>289</td>
</tr>
<tr>
<td>10.1.3</td>
<td>Internal Register and Flag Changes</td>
<td>289</td>
</tr>
<tr>
<td>10.1.4</td>
<td>Specifications</td>
<td>290</td>
</tr>
<tr>
<td>10.1.5</td>
<td>Note</td>
<td>290</td>
</tr>
<tr>
<td>10.1.6</td>
<td>Description</td>
<td>291</td>
</tr>
<tr>
<td>10.1.7</td>
<td>Flowchart</td>
<td>295</td>
</tr>
<tr>
<td>10.1.8</td>
<td>Program List</td>
<td>296</td>
</tr>
</tbody>
</table>
Section 1  Move Data

1.1  Set Constants

MCU:  H8/300 Series
       H8/300L Series

Label name:  FILL

1.1.1  Function

1. The software FILL places 1-byte constants in the data memory area.
2. The data memory area can have a free size.
3. Constants can have any length within the range 1 to 255 bytes.
4. This function is useful in initializing a RAM area.

1.1.2  Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Byte count (number of bytes)</td>
<td>R0L</td>
<td>1</td>
</tr>
<tr>
<td>Constants</td>
<td>R0H</td>
<td>1</td>
</tr>
<tr>
<td>Start address</td>
<td>R1</td>
<td>2</td>
</tr>
</tbody>
</table>

Output — — — — — —

1.1.3  Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0H</th>
<th>R0L</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

× : Unchanged
• : Indeterminate
‡ : Result
1.1.4 Specifications

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>10</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>3068</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

1.1.5 Note

The specified clock cycle count (3068) is for 255 bytes of constants.

1.1.6 Description

1. Details of functions
   a. The following arguments are used with the software FILL:
      R0L: Contains, as an input argument, the number of bytes to be placed in the data memory area holding constants.
      R0H: Contains, as an input argument, 1-byte constants to be placed in the data memory area.
      R1: Contains, as an input argument, the start address of the data memory area holding constants.
b. Figure 1.1 shows an example of the software FILL being executed.
When the input arguments are set as shown in (1), the constant H'34 set in R0H is placed in
the data memory area as shown in (2).

2. Notes on usage
   a. R0L is one byte long and should satisfy the relation H'01 ≤ R0L ≤ H'FF.
   b. Do not set "0" in R0L; otherwise, the software FILL can no longer be terminated.

3. Data memory
   The software FILL does not use the data memory.
4. Example of use

Set a constant, a byte count, and a start address in the arguments and call the software FILL as a subroutine.

```
WORK1 . DATA. B 0 ........... Reserves a data memory area (1 byte: contents=H'00) in which the user program places the number of bytes to be moved.
WORK2 . DATA. B 0 ........... Reserves a data memory area (1 byte: contents=H'00) in which the user program places constants.
WORK3 . RES. B 10 ........... Reserves a data memory area (10 bytes) that is set by the software FILL.
          MOV. B @WORK1, R0L ........... Places the number of bytes set by the user program in the R0L input argument.
          MOV. B @WORK2, R0H ........... Places the constants set by the user program in the R0H argument.
          MOV. W #WORK3, R1 ........... Places the start address of the data memory area allocated by the user program in the R1 argument.
          JSR @FILL .................. Calls the software FILL as a subroutine.
```

5. Operation

a. R1 is used as the pointer that indicates the address of the data memory area in which constants are placed.

b. The constants set in R0H in 16-bit absolute addressing mode are stored sequentially in the data memory area.

c. R0L is used as the pointer that indicates the number of bytes in the data memory area in which constants are placed. R0L is decremented each time a constant is placed in the data memory area until it reaches 0.
1.1.7 Flowchart

Flowchart:

- Fill
  - R0H → @R1
  - R1 + #1 → R1
  - R0L - #1 → R0L
  - R0L ≠ 0
    - Yes
      - RTS
    - No

Explanations:

- Places the constant (R0H) in the pointer that indicates the address (R1) in the data memory area.

- Increments R1.

- Decrements the counter (R0L) that indicates the number of bytes.

- Determines whether all specified constants have been set.
1.1.8 Program List

*** NR/300 ASSEMBLER       ** 08/18/92 11:04:12
PROGRAM NAME =

;******************************************************************************
;*   00 - NAME              :FILL OF CONSTANT DATA (FILL)
;******************************************************************************
;*   ENTRY           :R0L   (Byte counter)
;*                    R0H   (Constant data)
;*                    R1    (Start address)
;*   RETURN          :NOTHING
;******************************************************************************

FILL_cod C 0000                            .SECTION        FILL_code,CODE,ALIGN=2
FILL .EQU    $               ;Entry Point
MOV.B   R0H.@R1         ;Store constant data
ADDs.W  #1,R1           ;Increment address pointer
DEC.B   R0L             ;Decrement byte counter
BNE     FILL            ;Branch if Z flag = 0
RTS

.FILL_cod C 0000

;******************************************************************************
******TOTAL ERRORS       0
******TOTAL WARNINGS     0
1.2 Move Block 1

MCU: H8/300 Series
H8/300L Series

Label name: MOVE1

1.2.1 Function

1. The software MOVE1 moves block data from one data memory area to another.
2. The source and destination data memory areas can have a free size.
3. The block data can have any length within the range 1 to 255 bytes.

1.2.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Byte count (number of bytes)</td>
<td>R0L</td>
</tr>
<tr>
<td>Start address of source area</td>
<td>R1</td>
<td>2</td>
</tr>
<tr>
<td>Start address of destination area</td>
<td>R2</td>
<td>2</td>
</tr>
<tr>
<td>Output</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

1.2.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0H R0L R1 R2 R3 R4 R5 R6 R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×    ×     ×     ×     ×     ×     ×     ×     ×     ×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

× : Unchanged
• : Indeterminate
† : Result
1.2.4 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>14</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>4598</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

1.2.5 Note

The specified clock cycle count (4598) is for 255 bytes of block data.

1.2.6 Description

1. Details of functions
   a. The following arguments are used with the software MOVE1:
      R0L: Contains, as an input argument, the number of bytes of block data.
      R1: Contains, as an input argument, the start address of the source data memory area.
      R2: Contains, as an input argument, the start address of the destination data memory area.
b. Figure 1.2 shows an example of the software MOVE1 being executed. When the input arguments are set as shown in (1), the data is moved block by block from the source (H'FD80 to H'FD89) to the destination (H'FE80 to H'FE89) as shown in (2).
2. Notes on usage
   a. R0L is one byte long and should satisfy the relation H'01 \leq R0L \leq H'FF.
   b. Do not set "0" in R0L; otherwise, the software MOVE1 can no longer be terminated.
   c. Set the input arguments, ensuring that the source data memory area (A) does not overlap the destination data memory area (C) as shown in figure 1.3. In the case of figure 1.3, the overlapped block data (B) at the source is destroyed.

![Figure 1.3 Moving Block Data with Overlapped Data Memory Areas](image_url)

3. Data memory
   The software MOVE1 does not use the data memory.
4. Example of use

Set the start address of a source, the start address of a destination, and the number of bytes to be moved in the arguments and call the software MOVE1 as a subroutine.

| WORK1 | . DATA. B | 10 | Reserves a data memory area (1 byte: contents=H'0A) in which the user program places the number of bytes to be moved. |
|       | . ALIGN   | 2  | Places the data memory area (WORK1) at an even address. |
| WORK2 | . DATA. W | 0  | Reserves a data memory area (2 bytes: contents=H'0000) in which the user program places the start address of the source. |
| WORK3 | . DATA. W | 0  | Reserves a data memory area (2 bytes: contents=H'0000) in which the user program places the start address of the destination. |

```
WORK1 . DATA. B      10
.WORK2 . DATA. W     0
.WORK3

MOV. B @WORK1, R0L
MOV. W @WORK2, R1
MOV. W @WORK3, R2
JSR           @MOVE1
```

. ALIGN        2 Places the data memory area (WORK1) at an even address.

5. Operation

a. R1 is used as the pointer that indicates the address of the source and R2 the pointer that indicates the address of the destination.

b. The cycle of storing the data at the source in the work register (R0H) and then at the destination is repeated in 16-bit absolute addressing mode.

c. R0L is used as the counter that indicates the number of bytes moved. It is decremented each time 1-byte data is moved until it reaches 0.
1.2.7 Flowchart

MOVE1

@R1 → R0H

R0H → @R2

R1 + #1 → R1
R2 + #1 → R2

R0L - #1 → R0L

YES

R0L ≠ 0

NO

RTS

--- Places the data at the source in the work register (R0H).

--- Places R0H at the destination.

--- Increments the address pointer (R1) of the source and the address pointer (R2) of the destination.

--- Decrements the counter (R0L) that indicates the number of bytes to be moved.

--- Determines whether all specified data has been moved.
1.2.8 Program List

PROGRAM NAME =

;********************************************************************
;* 00-NAME :BROCK DATA TRANSFER (MOVE1)
;********************************************************************

6

7 ENTRY :R0L (Byte counter)
8 R1 (Source data start address)
9 R2 (Destination data start address)

11 RETURN :NOTHING

13 ;********************************************************************

15 MOVE1_code C 0000

16 .SECTION MOVE1_code,CODE,ALIGN=2

18 MOVE1 .EQU $ ;Entry point

20 MOV.L @R1,R0H ;Load source address data to R0H

22 MOV.L R0H,R2 ;Store R0H to destination address

24 ADD.L #1,R1 ;Increment source address pointer

26 ADD.L #1,R2 ;Increment destination address pointer

28 DEC R0L ;Decrement byte counter

30 BNE MOVE1 ;Branch if byte counter = 0

32 RTS

34 .END

*****TOTAL ERRORS       0
*****TOTAL WARNINGS     0
1.3  Move Block 2 (Example of the EEPMOV Instruction)

MCU:  H8/300 Series
      H8/300L Series

Label name:  MOVE2

1.3.1  Function

1.  The software MOVE2 moves block data from one data memory area to another.
2.  The source and destination data memory areas can have a free size.
3.  Data can be moved even where the source data memory area overlaps the destination data memory area.
4.  This is an example of the application software EEPMOV (move block instruction).

1.3.2  Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte count (number of bytes)</td>
<td>R4L</td>
<td>1</td>
</tr>
<tr>
<td>Start address of source area</td>
<td>R5</td>
<td>2</td>
</tr>
<tr>
<td>Start address of destination area</td>
<td>R6</td>
<td>2</td>
</tr>
<tr>
<td>Output</td>
<td>Error</td>
<td>C flag (CCR)</td>
</tr>
</tbody>
</table>

1.3.3  Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0H</th>
<th>R0L</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4H</th>
<th>R4L</th>
<th>R5</th>
<th>R6</th>
<th>R7</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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

×  : Unchanged
•  : Indeterminate
† : Result
### 1.3.4 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>58</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>1083</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

#### 1.3.5 Note

The specified clock cycle count (1083) is for 255 bytes of block data.

#### 1.3.6 Description

1. Details of functions
   a. The following arguments are used with the software MOVE2:
      R4L: Contains, as an input argument, the number of bytes of block data.
      R5: Contains, as an input argument, the start address of the source data memory area.
      R6: Contains, as an input argument, the start address of the destination data memory area.
      C flag (CCR): Determines the presence or absence of an error in the data length or address of the software MOVE2.
      C=0: All data has been moved.
      C=1: An input argument has an error.
b. Figure 1.4 shows an example of the software MOVE2 being executed. When the input arguments are set as shown in (1), the data is moved block by block from the source (H'FD80 to H'FD89) to the destination (H'FE80 to H'FE89) as shown in (2).
2. Notes on usage
   a. \text{R4L} is one byte long and should satisfy the relation \(H'01 \leq \text{R4L} \leq H'FF\).
   b. The source or destination data memory area must not extend over the end address (H'FFFF) to the start address (H'0000) as shown in figure 1.5; otherwise, the software MOVE2 fails.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure15.png}
\caption{Moving Block Data with Data Memory Area Extending over the Higher to Lower Addresses}
\end{figure}

3. Data memory
   The software MOVE2 does not use the data memory.
4. Example of use

Set the start address of a source, the start address of a destination, and the number of bytes to be moved in the arguments and call the software MOVE2 as a subroutine.

```
WORK1   . DATA. B 10 .......... Reserves a data memory area (1 byte: contents=H'0A) in which the user program places the number of bytes to be moved.
        . ALIGN 2 .......... Places the data memory area (WORK1) at an even address.
WORK2   . DATA. W 0 .......... Reserves a data memory area (2 bytes: contents=H'0000) in which the user program places the start address of the source.
WORK3   . DATA. W 0 .......... Reserves a data memory area (2 bytes: contents=H'0000) in which the user program places the start address of the destination.
        MOV. B @WORK1, R4L .......... Places the number of bytes set by the user program in the R0L argument.
        MOV. W @WORK2, R5 .......... Places the start address of the source set by the user program.
        MOV. W @WORK3, R6 .......... Places the start address of the destination set by the user program.
        JSR @MOVE2 .......... Calls the software MOVE2 as a subroutine.
```

5. Operation

a. R5 is used as the pointer that indicates the address of the source and R6 the pointer that indicates the address of the destination.
b. R4L is used as the counter that indicates the number of bytes moved. It is decremented each time 1-byte data is moved until it reaches 0.
c. When the input argument R4L is 0 or the start address of the source equals that of the destination, the C flag is set to 1 (error indicator) and the software MOVE2 terminates.
d. When the start address (B) of the destination data memory area is between the start address (A) and the end address (A + n – 1) of the source data memory area (A < B < A + n – 1; see figure 1.6), the data is moved sequentially from the higher address of the source in 16-bit absolute addressing mode.

![Figure 1.6](Image)

**Figure 1.6 Moving Data with Overlapped Data Memory Areas**

e. Except in the case of d., the EEPMOV instruction is used to move the data sequentially from the lower address.
1.3.7 Flowchart

MOVE2

R4L → R0L

#H'00 → R0H

R0L → R0L

3

EXIT
YES

NO

R0L = 0

YES

R5 = R6

NO

R0L - #1 → R0L

R5 → R2
R6 → R3

R2 + R0 → R2
R3 + R0 → R3

EP_MOV
YES

R6 > R2

NO

YES

R5 > R6

NO

NO

YES

Branches if the start address (R6) of the destination satisfies the condition R2 < R6 < R5.

Branches if the address pointer (R5) of the source equals the address pointer (R6) of the destination.

Decrement R0L.

Places the end address of the source data in R2 and the end address of the destination in R3.

Branches if the number of bytes is 0.

Branches if the address pointer (R5) of the source equals the address pointer (R6) of the destination.

Clears R0H.

Copies the number of bytes to be moved (R4L) to R0L.
Sets the C flag to 1.

Executes the block transfer instruction.

Clears the error indicator bit (C flag) to 0.

Decrements R4L.

Determines whether all specified data has been moved.

Moves 1-byte data from the source to the destination.

Decrements R2 and R3.

YES

NO

R4L ≠ 0

0 → C Flag

R2 - #1 → R2
R3 - #1 → R3
R4L - #1 → R4L

R2 - #1

R3 - #1

R4L - #1

B_MOV

EX1

EX2

EP_MOV

EX1

EXIT

EX2

EX1

EX2

1 → C Flag

Clears the error indicator bit (C flag) to 0.

Sets the C flag to 1.
1.3.8 Program List

*** H8/300 ASSEMBLER                VER 1.0B **   08/18/92 09:46:07

PROGRAM NAME =

1: ;********************************************************************
2: ;* 00 - NAME :BROCK DATA TRANSFER (MOVE2)
3: ;********************************************************************
4: ;* ENTRY :R4L (Byte counter)
5: ;* R5 (Source data start address)
6: ;* R6 (Destination data start address)
7: ;* RETURN :C bit of CCR (C=0;TRUE , C=1;FALSE)
8: ;*
9: ;********************************************************************
10: ;
11: ;
12: ;
13: ;
14: ;
15: MOVE2_co C 0000 .SECTION MOVE2_code,CODE,ALIGN=2
16: .EXPORT MOVE2
17: ;
18: MOVE2_co C 00000000 MOVE2 .EQU $ ;Entry point
19: MOVE2_co C 0000 CCE8 MOVE2 .EQU $ ;Entry point
20: MOVE2_co C 0002 F000 MOV.B #H'00,R0L
21: MOVE2_co C 0004 4CBE BEQ EXIT ;If byte counter="0" then exit
22: MOVE2_co C 0006 11C6 MOV.W R5,R5
23: MOVE2_co C 0008 0002 MOV.W #H'00,R0H
24: MOVE2_co C 000A 0000 MOV.W R1,R1
25: MOVE2_co C 000C 0208 MOV.W #H'80,R0L
26: MOVE2_co C 000E 0002 MOV.W R2,R2
27: MOVE2_co C 0010 0363 MOV.W #H'63,R2
28: MOVE2_co C 0012 0000 MOV.W R0,R0
29: MOVE2_co C 0014 0000 MOV.W R4,R4
30: MOVE2_co C 0016 0000 MOVE2 .EQU $ ;Entry point
31: MOVE2_co C 0018 4214 BHI EP_MOV ;Branch if R6>R2
32: MOVE2_co C 001A 4216 BHI EP_MOV ;Branch if R6>R2
33: MOVE2_co C 001C 4218 BHI EP_MOV ;Branch if R6>R2
34: MOVE2_co C 001E B_MOV
35: MOVE2_co C 0020 6834 MOV.B #H'84,R4H ;Load source data to R4H
36: MOVE2_co C 0022 6843 MOV.B #H'43,R4H ;Store R4H to destination
37: MOVE2_co C 0024 1802 SUBS.W #1,R2 ;Decrement source data pointer
38: MOVE2_co C 0026 1803 SUBS.W #1,R3 ;Decrement destination data pointer
39: MOVE2_co C 0028 180C MOV.W R4,R4
40: MOVE2_co C 002A 0410 ORC.B #H'01,CCR ;Set c flag for false
41: MOVE2_co C 002C 0401 ORC.B #H'01,CCR ;Set c flag for false
42: MOVE2_co C 002E RTS
43: MOVE2_co C 0030 46F4 BHI B_MOV ;Branch if R4L=0
44: MOVE2_co C 0032 06FE ANDC.S #H'FE,CCR ;Clear C flag of CCR
45: MOVE2_co C 0034 06FE ANDC.S #H'FE,CCR ;Clear C flag of CCR
46: MOVE2_co C 0036 0470 EX1
47: MOVE2_co C 0038 045F BFHMOV ;Clear C flag of CCR
48: MOVE2_co C 003A 045F BFHMOV ;Clear C flag of CCR
49: MOVE2_co C 003C 045F BFHMOV ;Clear C flag of CCR
50: MOVE2_co C 003E 045F BFHMOV ;Clear C flag of CCR
51: MOVE2_co C 003F 045F BFHMOV ;Clear C flag of CCR
52: ;
53: .END

*****TOTAL ERRORS       0
*****TOTAL WARNINGS     0
1.4 Move Character Strings

MCU: H8/300 Series
    H8/300L Series

Label name: MOVES

1.4.1 Function

1. The software MOVES moves character string block data from one data memory area to another.
2. When the delimiting H'00 appears in the block data, the software MOVES terminates.
3. The source or destination data memory area can have a free size.

1.4.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Start address of source area</td>
<td>R1</td>
</tr>
<tr>
<td></td>
<td>Start address of destination area</td>
<td>R2</td>
</tr>
<tr>
<td>Output</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

1.4.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0H</th>
<th>R0L</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

× : Unchanged
• : Indeterminate
† : Result
### 1.4.4 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>14</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>5116</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

### 1.4.5 Note

The specified clock cycle count (4598) is for 255 bytes of character string block data.
1.4.6 Description

1. Details of functions
   a. The following arguments are used with the software MOVES:
      R1: Contains, as an input argument, the start address of the source data memory area.
      R2: Contains, as an input argument, the start address of the destination data memory area.
   b. Figure 1.7 shows an example of the software MOVES being executed.
      When the input arguments are set as shown in (1), the data is moved block by block from
      the source (H'A000 to H'A009) to the destination (H'B000 to H'B009) as shown in (2).

![Diagram of software MOVES execution]

Figure 1.7 Example of Software MOVES Execution
2. Notes on usage
   a. Do not set H'00 in the source block data because H'00 is used as delimiting data; otherwise, the software MOVES terminates.
   b. Set input arguments, ensuring that the source data memory area (A) does not overlap the destination data memory area (C) as shown in figure 1.8. In the case of figure 1.8, the overlapped block data (B) at the source is destroyed.

![Figure 1.8 Moving Block Data with Overlapped Data Memory Areas](image)

3. Data memory
   The software MOVES does not use the data memory.
4. Example of use
Set the start address of a source and the start address of a destination in the arguments and call the software MOVES as a subroutine.

```
WORK1 . DATA. W 0 ........ Reserves a data memory area (2 bytes: contents=H'0000) in which the user program places the start address of the source.

WORK2 . DATA. W 0 ........ Reserves a data memory area (2 bytes: contents=H'0000) in which the user program places the start address of the destination.

.... MOV. W @WORK1, R1 ........ Places the start address of the source set by the user program.
MOV. W @WORK2, R2 ........ Places the start address of the destination set by the user program.

JSR @MOVES ........ Calls the software MOVES as a subroutine.
```

5. Operation
a. R1 is used as the pointer that indicates the address of the source and R2 the pointer that indicates the address of the destination.
b. The cycle of storing the data at the source in the work register (R0L) and then at the destination is repeated in 16-bit absolute addressing mode.
c. During the cycle, whether the R0L data is delimiting data is determined. If it is delimiting data (H'00), the software MOVES terminates; if not, moving the data continues.
1.4.7 Flowchart

MOVES

@R1 → R0L

R0L = 0

YES

R0L → @R2

R1 + #1 → R1
R2 + #1 → R2

EXIT

RTS

Places one byte of data at the source address indicated by the pointer (R1) in the work register (R0L).

Exits if the data is H'00.

Places the contents of R0L at the destination address indicated by the pointer (R2).

Increments R1 and R2.
1.4.8 Program List

*** R8/300 ASSEMBLER VER 1.0B ** 08/18/92 09:46:36
PROGRAM NAME =

1 ;*******************************************************************************
2 ;*
3 ;* 00 - NAME :TRANSFER OF STRING (MOVES)
4 ;*
5 ;*******************************************************************************
6 ;*
7 ;* ENTRY :R1 (Source address)
8 ;* R2 (Destination address)
9 ;*
10 ;* RETURN :NOTHING
11 ;*
12 ;*******************************************************************************
13 ;*
14 MOVES_co C 0000 .SECTION MOVES_code,CODE,ALIGN=1
15 .EXPORT MOVES
16 ;
17 MOVES_co C 00000000 MOVES .EQU $ ;Entry point
18 MOVES_co C 0000 4818 MOV.B @R1,R0L ;Load source address data to R0L
19 MOVES_co C 0002 4708 SEQ EXIT ;Branch if R0H = R0L
20 MOVES_co C 0004 68A8 MOV.B R0L,#R2 ;Store R0L to destination address
21 MOVES_co C 0006 0601 ADDS.W #1,R1 ;Increment source address pointer
22 MOVES_co C 0008 0602 ADDS.W #1,R2 ;Increment destination address pointer
23 MOVES_co C 000A 40F4 BRA MOVES ;Branch always
24 ;
25 MOVES_co C 00DC EXIT RTS
26 MOVES_co C 00DC 7A70 .END
27 ;
28 ****TOTAL ERRORS 0
29 ****TOTAL WARNINGS 0
Section 2  Branch by Table

2.1  Branch by Table

MCU:  H8/300 Series
       H8/300L Series

Label name:  CCASE

2.1.1  Function

1. The software CCASE determines the start address of a processing routine for a 1-word (2-byte) command.
2. This function is useful in decoding data input from the keyboard or performing a process appropriate for input data.

2.1.2  Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Command</td>
<td>R0</td>
<td>2</td>
</tr>
<tr>
<td>Start address of data table</td>
<td>R1</td>
<td>2</td>
</tr>
<tr>
<td>Output Start address of processing routine</td>
<td>R4</td>
<td>2</td>
</tr>
<tr>
<td>Command</td>
<td>C flag (CCR)</td>
<td></td>
</tr>
</tbody>
</table>

2.1.3  Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>•</td>
<td>†</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>•</td>
<td>×</td>
<td>•</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>†</td>
</tr>
</tbody>
</table>

×  : Unchanged
•  : Indeterminate
†  : Result
### 2.1.4 Specifications

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>28</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>74</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

### 2.1.5 Note

The specified clock cycle count (74) is for the example of figure 2.1 being executed.

### 2.1.6 Description

1. Details of functions
   a. The following arguments are used with the software CCASE:
      R0: Contains, as an input argument, 2-byte commands.
      R1: Contains, as an input argument, the start address of a data table storing the command (R0) and the start address of a processing routine.
      R4: Contains, as an output argument, the start address (2 bytes) of the processing routine for the command (R0).
      C flag (CCR): Determines the state of data after the software CCASE has been executed.
      C flag = 1: The data matching the command set in R0 was on the data table.
      C flag = 0: The data matching the command set in R0 was not on the data table.
b. Figure 2.1 shows an example of the software CCASE being executed. When the input arguments are set as shown in (1), the program refers to the data table (see figure 2.1 and places the start address of the processing routine in R4 as shown in (2).

c. Executing the software CCASE requires a data table as shown in figure 2.1. The data table is as follows:

(i) The table contains data groups each consisting of 4 bytes (2 words) beginning with address H'FD80 and delimiting data H'0000 indicating the end of the table.

(ii) The first word of each data group (2 words) contains a command and the second word contains the start address of the processing routine in the order of the upper bytes followed by the lower bytes.

![Figure 2.1 Example of Software CCASE Execution](image)

- (1) Input arguments
  - R0(H'0042) 0 0 4 3
  - R1(H'FD80) F D 8 0

- (2) Output arguments
  - R4(H'F200) F 2 0 0
  - C flag (CCR) 1
2. Notes on usage
   Do not use H'0000 as a command in the data table because H'0000 is used as delimiting data.
3. Data memory
   The software CCASE does not use the data memory.
4. Example of use
Set commands and the start address of the data table in the arguments and call the software CCASE as a subroutine.

```
WORK1  .RES.B 1

MOV. W #DTABLE, R1
MOV. B @WORK1, R0L
JSR           @CCASE
RES.B     1
```

Reserves a data memory area in which the user program places a command.

Places in the input argument the start address of the data table set by the user program.

Places in the argument the command stored by the user program.

Calls the software CCASE as a subroutine.

Branches when there is no data that matches the command on the data table.

```
JSR           ERROR
Bcc ERROR
```

Program to be branched to processing routine

```
.DECISION D-TABLE, DATA, ALIGN=2

DTABLE .DATA.W H'0041  Command at "0A"
        .DATA.W H'FF00  Start address of processing routine for command at "0A"
        .DATA.W H'0042  Command at "0B"
        .DATA.W H'FF100 Start address of processing routine for command at "0B"
        .DATA.W H'0043  Command at "0C"
        .DATA.W H'FF200 Start address of processing routine for command at "0C2"
        .DATA.W H'0000  Delimiting data

Note  Example of program to be branched to processing routine
```
The software CCASE merely places the start address of a processing routine in R4. Branching to a processing routine requires the following program:

```
JSR           @CCASE

······ Calls CCASE as a subroutine.
·········· If the C flag is 0, operation branches to an error program.
··········· Calls the processing program as a subroutine.

Bcc  ERROR
```

Branching to processing routine

```
JSR           @R4
```

ERROR  Error program

5. Operation
   a. R1 is used as the pointer that indicates the address of the data table.
   b. The commands are read sequentially from the start address of the data table in register indirect addressing mode. Then the contents of each command on the data table are compared with those of each input command (R0).
   c. When a command on the table matches R0, the start address of the processing routine placed at the address next to the command is set in R4. Then the C flag is set to 1 and the software CCASE terminates.
   d. When the command on the data table is H'0000, the C flag is cleared to 0 and the software CCASE terminates.
2.1.7 Flowchart

- **CCASE**
  - \#H'0004 \(\rightarrow\) R6
    - **Lbl**
  - \@R1 \(\rightarrow\) R2
    - **R2 = 0**
      - **YES**
        - **R0 = R2**
          - **YES**
            - **R1 + R6 \(\rightarrow\) R1**
          - **NO**
            - **NO**
              - **R1 + R6 \(\rightarrow\) R1**
    - **NO**
      - **YES**
        - **R2 = 0**
          - **YES**
            - **@\(2, R1\) \(\rightarrow\) R4**
            - **1 \(\rightarrow\) C Flag**
          - **NO**
            - **NO**
              - **R1 + R6 \(\rightarrow\) R1**

- **EX1**
  - **RTS**
    - **FALSE**
      - **0 \(\rightarrow\) C Flag**
### Program List

**2.1.8 Program List**

```
1                           ;*****************************************************************************
2                           ;* 00 - NAME :TABLE BRANCH (CCASE)
3                           ;*****************************************************************************
4                           ;* ENTRY :R0 COMMAND
5                           ;* R1 DATA TABLE START ADDRESS
6                           ;* RETURN :R4 MODULE START ADDRESS
7                           ;* C bit of CCR C=1;TRUE , C=0;FALSE
8                           ;*****************************************************************************
9                           ;*ENTRY :R0 COMMAND
10                          ;* R1 DATA TABLE START ADDRESS
11                          ;* RETURN :R4 MODULE START ADDRESS
12                          ;* C bit of CCR C=1;TRUE , C=0;FALSE
13                          ;*****************************************************************************
14                          ;
15 CASE_code C 0000         .SECTION CASE_code,CODE,ALIGN=2
16                          .EXPORT CASE
17                          ;
18 CASE_code C 00000000      CASE .EQU $ ;Entry point
19 CASE_code C 0000 79060004 MOV.W #H'0004,R6
20 CASE_code C 0004          LBL
21 CASE_code C 0004 6912      MOV.W #R1,R2
22 CASE_code C 0004 4710      BEQ FALSE ;If table "END" then exit
23 CASE_code C 0008 1002      CMP.W R0,R2 ;Branch if command find
24 CASE_code C 000A 4704      BRQ TRUE ;Branch if command find
25 CASE_code C 000C 0961      ADD.W R6,R1 ;Increment table address
26 CASE_code C 000E 40F4      BRA LBL ;Branch always
27 CASE_code C 0010          TRUE
28 CASE_code C 0010 6F140002  MOV.W @(H'2,R1),R4 ;Load module start address
29 CASE_code C 0014 0401      SRC #H'01,CCR ;Set C flag for true
30 CASE_code C 0018          EXI
31 CASE_code C 0018 5670      RTS
32 CASE_code C 0018          FALSE
33 CASE_code C 0018 06FE      ANDC #H'FE,CCR ;Clear C flag for false
34 CASE_code C 001A 40FA      BRA EXI
35 ;
36 .END
```

*TOTAL ERRORS* 0

*TOTAL WARNINGS* 0
Section 3  ASCII CODE PROCESSING

3.1  Change ASCII Code from Lowercase to Uppercase

MCU:  H8/300 Series
       H8/300L Series

Label name:  TPR

3.1.1  Function

1. The software TPR changes a lowercase ASCII code to a corresponding uppercase ASCII code.
2. All data used with the software TPR is ASCII code.

3.1.2  Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Lowercase ASCII code</td>
<td>R0L</td>
</tr>
<tr>
<td>Output</td>
<td>Uppercase ASCII code</td>
<td>R0L</td>
</tr>
</tbody>
</table>

3.1.3  Internal Register and Flag Changes

R0H  R0L  R1   R2   R3   R4   R5   R6   R7
×    †   •   •   •   •   •   •   •

I  U  H  U  N  Z  V  C
•  •  ×  •  ×  ×  ×  ×

× : Unchanged
• : Indeterminate
† : Result
### 3.1.4 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>14</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>24</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>
3.1.5 Description

1. Details of functions
   a. The following argument is used with the software TPR:
      R0L: Contains, as an input argument, a lowercase ASCII code. After execution of the
      software TPR, the corresponding uppercase ASCII code is placed in R0L.
   b. Figure 3.1 shows an example of the software TPR being executed. When the lowercase
      ASCII code 'a' (H'61) is set as shown in (1), it is converted to the uppercase ASCII code 'A'
      (H'41), which then is placed in R0L as shown in (2).

![Figure 3.1 Example of Software TPR Execution](image)

2. Notes on usage
   R0L must contain a lowercase ASCII code. If any other code is placed in R0L, the input data is
   retained in R0L.

3. Data memory
   The software TPR does not use the data memory.
4. Example of use

Set a lowercase ASCII code in the input argument and call the software TPR as a subroutine.

<table>
<thead>
<tr>
<th>WORK1</th>
<th>RES. B 1</th>
<th>Reserves a data memory area in which the user program places a lowercase ASCII code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORK2</td>
<td>RES. B 1</td>
<td>Reserves a data memory area in which a corresponding uppercase ASCII code is placed in the user program.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Places the lowercase ASCII code set by the user program in the input argument.</td>
</tr>
<tr>
<td>MOV. B</td>
<td>@WORK1, R0L</td>
<td>Calls the software TPR as a subroutine.</td>
</tr>
<tr>
<td>JSR</td>
<td>@TPR</td>
<td>Places the uppercase ASCII code set in the output argument in the data memory area of the user program.</td>
</tr>
<tr>
<td>MOV. B</td>
<td>R0, @WORK2</td>
<td></td>
</tr>
</tbody>
</table>

5. Operation

a. compare instruction (CMP.B) is used to determine whether the input data set in R0L is a lowercase ASCII code.

b. H’20 is subtracted from the lowercase ASCII code to obtain a corresponding uppercase ASCII code.

c. If the input data is not a lowercase ASCII code, the program retains the input data and terminates processing.
3.1.6 Flowchart

Subtracts #H'20 from R0L to convert the lowercase code to a corresponding uppercase code.

Branches when the value set in R0L is any code other than the lowercase ASCII code 'a' to 'z' (#H'61 to #H'7A).

Subtracts #H'20 from R0L to convert the lowercase code to a corresponding uppercase code.
3.1.7 Program List

*** H8/300 ASSEMBLER                VER 1.0B **   08/18/92 09:47:44
PROGRAM NAME =

1 ;********************************************************************
2 ;* 00 - NAME :CHANGE ASCII CODE LOWERCASE
3 ;* TO UPPERCASE (TPR)
4 ;********************************************************************
5 ;*
6 ;********************************************************************
7 ;*
8 ;* ENTRY :R0L (ASCII CODE LOWERCASE)
9 ;*
10 ;* RETURN :R0L (ASCII CODE UPPERCASE)
11 ;*
12 ;********************************************************************
13 ;
14 TPR_code C 0000 .SECTION TPR_code,CODE,ALIGN=1
15 .EXPORT TPR
16 ;
17 TPR_code C 00000000 TPR .EQU $ ;Entry point
18 TPR_code C 0000 8A62 CMP.B #$'61,R0L ;Branch if R0L<#$'60
19 TPR_code C 0002 4508 BCS EXIT ;Branch if R0L<#$'60
20 TPR_code C 0004 A87A CMP.B #$'7A,R0L ;Branch if R0L>#$'7B
21 TPR_code C 0006 4204 BHI EXIT ;Branch if R0L>#$'7B
22 TPR_code C 0008 F020 MOV.B #$'20,R0H ;Lowercase - #$'20 -> Uppercase
23 TPR_code C 000A 1808 SUB.B R0H,R0L ;Lowercase - #$'20 -> Uppercase
24 TPR_code C 000C EXIT
25 TPR_code C 000C 5470 RTS
26 ;
27 ;.END

*****TOTAL ERRORS   0
*****TOTAL WARNINGS 0
3.2 Change an ASCII Code to a 1-Byte Hexadecimal Number

MCU:  H8/300 Series
       H8/300L Series

Label name:  NIBBLE

3.2.1 Function

1. The software NIBBLE changes an ASCII code (‘0’ to ‘9’ and ‘A’ to ‘F’) to a corresponding 1-byte hexadecimal number.
2. All data used with the software NIBBLE is ASCII code.

3.2.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input ASCII code</td>
<td>R0L</td>
<td>1</td>
</tr>
<tr>
<td>Output 1-byte hexadecimal number</td>
<td>R0L</td>
<td>1</td>
</tr>
<tr>
<td>Convert or not</td>
<td>C flag (CCR)</td>
<td></td>
</tr>
</tbody>
</table>

3.2.3 Internal Register and Flag Changes

```
R0H  R0L  R1  R2  R3  R4  R5  R6  R7
•   †   •   •   •   •   •   •   •

I  U    H  U   N  Z  V   C
•   •   ×   •   ×   ×   ×   ×
```

×: Unchanged
*: Indeterminate
†: Result
### 3.2.4 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>24</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>38</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>
3.2.5 Description

1. Details of functions
   a. The following arguments are used with the software NIBBLE:
      R0L: Contains, as an input argument, an ASCII code. After execution of the software NIBBLE, the corresponding 1-byte hexadecimal number is placed in R0L.
      C flag (CCR): Holds, as an output argument, the state of ASCII code after execution of the software NIBBLE.
      C flag = 1: The input ASCII code is any other than '0' to '9' or 'A' to 'F'.
      C flag = 0: The input ASCII code is '0' to '9' or 'A' to 'F'.
   b. Figure 3.2 shows an example of the software NIBBLE being executed. When the input argument is set as shown in (1), a corresponding 1-byte hexadecimal number (H'0F) is placed in R0L as shown in (2).

<table>
<thead>
<tr>
<th>(1) Input argument</th>
<th>(2) Output argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0L</td>
<td>R0L</td>
</tr>
<tr>
<td>ASCII code 'F'</td>
<td>1-byte hexadecimal</td>
</tr>
<tr>
<td>H'46</td>
<td>H'0F</td>
</tr>
<tr>
<td></td>
<td>C flag = 0</td>
</tr>
</tbody>
</table>

   Figure 3.2 Example of Software NIBBLE Execution

2. Notes on usage
   If any data other than ASCII code '0' to '9' or 'A' to 'F' is set in R0L, the data is destroyed after execution of the software NIBBLE.

3. Data memory
   The software NIBBLE does not use the data memory.
### 4. Example of use

Set an ASCII code in the input argument and call the software NIBBLE as a subroutine.

<table>
<thead>
<tr>
<th>WORK1</th>
<th>.RES. B 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV. B @WORK1, R0L</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WORK2</th>
<th>.RES. B 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV. B R0L, @WORK2,</td>
<td></td>
</tr>
</tbody>
</table>

| JSR @NIBBLE |
| BCS SKIP |

| SKIP |
| Processing routine for invalid ASCII code |

- Reserves a data memory area in which the user program places a 1-byte ASCII code.
- Reserves a data memory area in which the user program places a corresponding 1-byte hexadecimal.
- Places the ASCII code set by the user program in the input argument.
- Calls the software NIBBLE as a subroutine.
- Branches to the skip processing routine if the input data is any other than '0' to '9' or 'A' to 'F'.
- Places the 1-byte hexadecimal set in the output argument in the data memory area of the user program.
5. Operation
   a. On the basis of the status of the C flag showing the result of operations, the software
      NIBBLE determines whether the data set in R0L falls in the ‘0’ to ‘F’ range of the ASCII
      code table (\[\] in table 3.1).
   b. The software further perform operations to delete the ‘:’ to ‘@’ range ([ ] in table
      3.1).
   c. If the input data is outside the ‘0’ to ‘9’ and ‘A’ to ‘F’ ranges, 1 is set in the C flag in the
      processes a. and b..

Table 3.1  ASCII Code Table

<table>
<thead>
<tr>
<th>LSD</th>
<th>MSD 0</th>
<th>MSD 1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NUL</td>
<td>DLE</td>
<td>SP</td>
<td>0</td>
<td>P</td>
<td>`</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>SOH</td>
<td>DC1</td>
<td>!</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
</tr>
<tr>
<td>0010</td>
<td>STX</td>
<td>DC2</td>
<td>*</td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
</tr>
<tr>
<td>0011</td>
<td>ETX</td>
<td>DC3</td>
<td>#</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
</tr>
<tr>
<td>0100</td>
<td>EOT</td>
<td>DC4</td>
<td>$</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
</tr>
<tr>
<td>0101</td>
<td>ENG</td>
<td>NAK</td>
<td>%</td>
<td>5</td>
<td>E</td>
<td>U</td>
<td>e</td>
<td>u</td>
</tr>
<tr>
<td>0110</td>
<td>ACK</td>
<td>SYN</td>
<td>&amp;</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
</tr>
<tr>
<td>0111</td>
<td>BEL</td>
<td>ETB</td>
<td>'</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
</tr>
<tr>
<td>1000</td>
<td>BS</td>
<td>CAN</td>
<td>(</td>
<td>8</td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
</tr>
<tr>
<td>1001</td>
<td>HT</td>
<td>EM</td>
<td>)</td>
<td>9</td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
</tr>
<tr>
<td>1010</td>
<td>LF</td>
<td>SUB</td>
<td>*</td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>VT</td>
<td>ESC</td>
<td>+</td>
<td>K</td>
<td>[</td>
<td>k</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>FF</td>
<td>FS</td>
<td>,</td>
<td>L</td>
<td>\</td>
<td>l</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>CR</td>
<td>GS</td>
<td>&lt;</td>
<td>M</td>
<td>)</td>
<td>m</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>SO</td>
<td>RS</td>
<td>&gt;</td>
<td>N</td>
<td>\</td>
<td>n</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>SI</td>
<td>VS</td>
<td>/</td>
<td>O</td>
<td>←</td>
<td>o</td>
<td>DEL</td>
<td></td>
</tr>
</tbody>
</table>

RENESAS
3.2.6 Flowchart

- **NIBBLE**

  - **R0L < '0'**
    - NO
      - **R0L - H'30 → R0**
    - YES
      - **R0L + H'E9 → R0L**

  - **C flag = 1**
    - NO
      - **R0L + H'06 → R0L**
    - YES
      - **R0L + H'07 → R0L**

  - **C flag = 1**
    - NO
      - **R0L + H'0A → R0L**
    - YES
      - **0 → C flag**

- **EXIT**

  - **RTS**

Branches to the processing routine if the input argument (R0L) is less than '0' of ASCII code.

Branches to the processing routine, with the C flag set, if R0L is not less than 'G' of ASCII code.

Branches to the processing routine, with the C flag set, if R0L is not less than 'A' of ASCII code.

Branches to the processing routine, with the C flag set, if R0L is between ':' and '@'.

Changes the value of R0L to a hexadecimal (H'00 to H'0F).

Clears the bit indicating the state of end (the C flag) to 0, indicating the input data has been changed.
3.2.7 Program List

*** 88/300 ASSEMBLER            VER 1.0B  **  08/18/92 20:08:15

PROGRAM NAME =

1  ;*******************************************************************************
2  ;*
3  ;* 00 - NAME :CHANGE 1 BYTE ASCII CODE
4  ;* TO 4 BIT HEXAGON (NIBBLE)
5  ;*
6  ;*******************************************************************************
7  ;*
8  ;* ENTRY :R0L (1 BYTE ASCII CODE)
9  ;*
10  ;* RETURN :R0L (4 BIT HEXADECIMAL)  C flag of CCR  (C=0;FALSE , C=1;TRUE)
11  ;*
12  ;*******************************************************************************
13  ;*
14  ;*******************************************************************************
15 NIBBLE_c C 0000 .SECTION NIBBLE_code,CODE,ALIGN=2
16 .EXPORT NIBBLE
17  
18 NIBBLE_c C 0000 NIBBLE
19 NIBBLE_c C 0000 F030 MOV.B #H'30,R0H
20 NIBBLE_c C 0002 1008 SUB.B R0H,R0L ;R0L - #H'30 -> R0L
21 NIBBLE_c C 0004 4510 BCS EXIT ;Branch if R0L<'0'
22 NIBBLE_c C 0006 4508 ADD.B #H'07,R0L
23 NIBBLE_c C 0008 450C BCS EXIT ;Branch if R0L<'F'
24 NIBBLE_c C 000A 8806 ADD.B #H'06,R0L
25 NIBBLE_c C 000C 4504 BCS LBL ;Branch if R0L<=H'FF
26 NIBBLE_c C 000E 4504 ADD.B #H'07,R0L
27 NIBBLE_c C 0010 4504 BCS EXIT ;Branch if R0L<=H'FF
28 NIBBLE_c C 0012 LBL
29 NIBBLE_c C 0012 880A ADD.B #H'0A,R0L ;Change R0L to ASCII CODE
30 NIBBLE_c C 0014 06FE ANDC #H'FE,CCR ;Clear C flag of CCR
31 NIBBLE_c C 0016 EXIT
32 NIBBLE_c C 0016 5470 RTS
33  
34 .END

*****TOTAL ERRORS  0
*****TOTAL WARNINGS  0
3.3 Change an 8-Bit Binary Number to a 2-Byte ASCII Code

MCU: H8/300 Series
H8/300L Series

Label name: COBYTE

3.3.1 Function

1. The software COBYTE changes an 8-bit binary number to a corresponding 2-byte ASCII code.
2. All data used with the software COBYTE is ASCII code.

3.3.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>R0L</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>R1</td>
<td>2</td>
</tr>
</tbody>
</table>

3.3.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0H</th>
<th>R0L</th>
<th>R1</th>
<th>R2H</th>
<th>R2L</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

• : Unchanged
† : Result
: Indeterminate
3.3.4 Specifications

<table>
<thead>
<tr>
<th>Specifications</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>38</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>72</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

3.3.5 Description

1. Details of functions
   a. The following arguments are used with the software COBYTE:
      R0L: Contains, as an input argument, an 8-bit binary number to be changed to a corresponding ASCII code.
      R1: Contains, as an output argument, 1-byte ASCII code data in the upper 4 bits and the lower 4 bits each of the 8-bit binary number.
   b. Figure 8-1 shows an example of the software COBYTE being executed. When the input argument is set as shown in á@, 2-byte ASCII code data is placed in R1 as shown in áA.
2. Notes on usage
    The 8-bit binary number set in R0L is destroyed after execution of the software COBYTE.

3. Data memory
    The software COBYTE does not use the data memory.

4. Example of use
    Set an 8-bit binary number in the input argument and call the software COBYTE as a subroutine.

<table>
<thead>
<tr>
<th>WORK1</th>
<th>.RES. B 1</th>
<th>Reserves a data memory area in which the user program places an 8-bit binary number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ALIGN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WORK2</td>
<td>.RES. W 1</td>
<td>Reserves a data memory area in which the user program places a corresponding 2-byte ASCII code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV. B @WORK1, R0L</td>
<td>Places the 8-bit binary number set by the user program in R0L.</td>
<td></td>
</tr>
<tr>
<td>JSR @COBYTE</td>
<td>Calls the software COBYTE as a subroutine.</td>
<td></td>
</tr>
<tr>
<td>MOV. W R1, @WORK2</td>
<td>Places the 2-byte ASCII code set in the output argument in the data memory area of the user program.</td>
<td></td>
</tr>
</tbody>
</table>
5. Operation
a. The 8-bit binary number is separated into two bit groups, the upper 4 bits and the lower 4 bits.
b. A compare instruction is used to determine whether the data (the upper 4 bits + the lower 4 bits) is in the H'00 to H'09 range ([ ] in table 3.2) or in the H'0A to H'0F range ([ ] in table 3.2). H'30 is added to the data if it falls in the H'00 to H'09 range and H'37 to the data if it falls in the H'0A to H'0F range for change to a corresponding ASCII code.

Table 3.2  ASCII Code Table

<table>
<thead>
<tr>
<th></th>
<th>MSD</th>
<th>0 0 0</th>
<th>0 0 1</th>
<th>0 1 0</th>
<th>0 1 1</th>
<th>1 0 0</th>
<th>1 0 1</th>
<th>1 1 0</th>
<th>1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSD</td>
<td>0</td>
<td>0 0 0</td>
<td>0 0 1</td>
<td>0 1 0</td>
<td>0 1 1</td>
<td>1 0 0</td>
<td>1 0 1</td>
<td>1 1 0</td>
<td>1 1 1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>NUL</td>
<td>DLE</td>
<td>SP</td>
<td>0</td>
<td>@</td>
<td>P</td>
<td>`</td>
<td>p</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>SOH</td>
<td>DC1</td>
<td>!</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>STX</td>
<td>DC2</td>
<td>*</td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>ETX</td>
<td>DC3</td>
<td>#</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>EOT</td>
<td>DC4</td>
<td>$</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>ENG</td>
<td>NAK</td>
<td>%</td>
<td>5</td>
<td>E</td>
<td>U</td>
<td>e</td>
<td>u</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>ACK</td>
<td>SYN</td>
<td>&amp;</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>BEL</td>
<td>ETB</td>
<td>'</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>BS</td>
<td>CAN</td>
<td>(</td>
<td>8</td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>HT</td>
<td>EM</td>
<td>)</td>
<td>9</td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>LF</td>
<td>SUB</td>
<td>:</td>
<td>:</td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>VT</td>
<td>ESC</td>
<td>+</td>
<td>;</td>
<td>K</td>
<td>`</td>
<td>k</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>FF</td>
<td>FS</td>
<td>,</td>
<td>&lt;</td>
<td>L</td>
<td>\</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>CR</td>
<td>GS</td>
<td>-</td>
<td>=</td>
<td>M</td>
<td>)</td>
<td>m</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>SO</td>
<td>RS</td>
<td>*</td>
<td>&gt;</td>
<td>N</td>
<td>↑</td>
<td>n</td>
<td>~</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>SI</td>
<td>VS</td>
<td>/</td>
<td>?</td>
<td>O</td>
<td>←</td>
<td>o</td>
<td>DEL</td>
</tr>
</tbody>
</table>

RENESAS
3.3.6 Flowchart

- **COBYTE**
  - R0L → R0H
  - Shift R0H 4 bits right
  - R0L ^ #H'0F → R0L
  - R0H → R2L
  - CONIB
  - R2L → R1H
  - R0L → R2L
  - CONIB
  - R2L → R1L
  - RTS

- **CONIB**
  - YES
  - R2L ≥ #H'0A
  - NO
  - R2L + #H'30 → R2L
  - RTS

  - LBL
  - R2L + #H'37 → R2L
  - RTS

- **COBYTE**
  - Copies the 8-bit binary placed in R0L to R0H.
  - Moves the upper 4 bits of R0H to the lower 4 bits.
  - Clears the upper 4 bits of the 8-bit binary set in R0L.

- **CONIB**
  - Calls the software CONIB as a subroutine to change the upper 4 bits of the 8-bit binary to ASCII code.

- **CONIB**
  - Calls the software CONIB as a subroutine to change the lower 4 bits of the 8-bit binary to ASCII code.

- **YES**
  - Branches if the 4-bit binary is not less than #H'0A.

- **NO**
  - Adds #H'30 to R2L to change hexadecimal number 0-9 to corresponding ASCII code '0'-'9'.

- **R2L + #H'30 → R2L**
  - Adds #H'37 to R2L to convert hexadecimal A-F to ASCII code 'A'-'F'.

- **RTS**

---

**SHIFT R0H 4 Bits Right**

- Copies the 8-bit binary placed in R0L to R0H.
- Moves the upper 4 bits of R0H to the lower 4 bits.
- Clears the upper 4 bits of the 8-bit binary set in R0L.

**CONIB**

- Calls the software CONIB as a subroutine to change the upper 4 bits of the 8-bit binary to ASCII code.
- Calls the software CONIB as a subroutine to change the lower 4 bits of the 8-bit binary to ASCII code.

**YES**

- Branches if the 4-bit binary is not less than #H'0A.

**NO**

- Adds #H'30 to R2L to change hexadecimal number 0-9 to corresponding ASCII code '0'-'9'.

**R2L + #H'30 → R2L**

- Adds #H'37 to R2L to convert hexadecimal A-F to ASCII code 'A'-'F'.

---

**RENAEXEC**
### 3.3.7 Program List

---

**Program Name:**

1

```
1                                      ;********************************************************************
2                                      ;*
3                                      ;* 00 - NAME :CHANGE 1 BYTE HEXADECIMAL
4                                      ;* TO 2 BYTE ASCII CODE (COBYTE)
5                                      ;*
6                                      ;********************************************************************
7                                      ;*
8                                      ;* ENTRY :ROL (1 BYTE HEXADECIMAL)
9                                      ;*
10                                     ;* RETURN :R1 (2 BYTE ASCII CODE)
11                                     ;*
12                                     ;********************************************************************
13                                     ;
14 COBYTE_c C 0000                      .SECTION COBYTE_code,CODE,ALIGN=2
15                                      .EXPORT COBYTE
16                                      ;
17 COBYTE_c C 0000 00000000             COBYTE .EQU $ ;Entry Point
18 COBYTE_c C 0000 0000                 MOV.B R0L,R0H
19                                      ;
20 COBYTE_c C 0002 1100                 SHLR R0H
21 COBYTE_c C 0004 1100                 SHLR R0H
22 COBYTE_c C 0006 1100                 SHLR R0H
23 COBYTE_c C 0008 1100                 SHLR R0H ;Select upper 4 bit hexadecimal(R0H)
24                                      ;
25 COBYTE_c C 000A E80F                 AND.B #H'0F,R0L ;Select lower 4 bit hexadecimal(R0L)
26                                      ;
27 COBYTE_c C 000C 0C04                 MOV.B R0L,R2L
28 COBYTE_c C 000E 5504                 BSR CONIB ;Branch subroutine CONIB
29 COBYTE_c C 0010 0CA1                 MOV.B R2L,R1H ;Set 1st ASCII code to R1H
30                                      ;
31 COBYTE_c C 0012 0C0A                 MOV.B R1L,R2L
32 COBYTE_c C 0014 5504                 BSR CONIB ;Branch subroutine CONIB
33 COBYTE_c C 0016 0CA1                 MOV.B R2L,R1L ;Set 2nd ASCII code to R1L
34                                      ;
35 COBYTE_c C 0018 5470                 RTS
36                                      ;--------------------------------------------------------------------
37 COBYTE_c C 001A 00000000             COBYTE .EQU 5 ;Entry Point
38                                      ;
39 COBYTE_c C 001A 0000 00000000         MOV.B R0L,R0H
40                                      ;
41 COBYTE_c C 001A 0002 1100             SHLR R0H
42 COBYTE_c C 001A 0004 1100             SHLR R0H
43 COBYTE_c C 001A 0006 1100             SHLR R0H
44 COBYTE_c C 001A 0008 1100             SHLR R0H ;Select upper 4 bit hexadecimal(R0H)
45                                      ;
46 COBYTE_c C 001A 000A E80F             AND.B #H'0F,R0L ;Select lower 4 bit hexadecimal(R0L)
47                                      ;
48 COBYTE_c C 001A 000C 0C04             MOV.B R0L,R2L
49 COBYTE_c C 001A 000E 5504             BSR CONIB ;Branch subroutine CONIB
50 COBYTE_c C 001A 0010 0CA1             MOV.B R2L,R1H ;Set 1st ASCII code to R1H
51                                      ;
52 COBYTE_c C 001A 0012 0C0A             MOV.B R1L,R2L
53 COBYTE_c C 001A 0014 5504             BSR CONIB ;Branch subroutine CONIB
54 COBYTE_c C 001A 0016 0CA1             MOV.B R2L,R1L ;Set 2nd ASCII code to R1L
55                                      ;
56 COBYTE_c C 001A 0018 5470             RTS
57                                      ;--------------------------------------------------------------------
58 CONIB                                ;Change R2L to ASCII code
59                                        CMP.B #H'0A,R2L
60                                        SBC LBL ;Branch if R2L ASCII "A"-"F"
61                                       ;
62 CONIB                                ;Rshape R2L ASCII "0"- "9"
63                                        ADD.B #H'30,R2L
64                                        RTS
65                                      ;
66                                      .END
```

---

**Total Errors:** 0

**Total Warnings:** 0

---

**Renesas**
Section 4  BIT PROCESSING

4.1 Count the Number of Logic 1 Bits in 8-Bit Data (HCNT)

MCU:  H8/300 Series
       H8/300L Series

Label name:  HCNT

4.1.1 Function

1. The software HCNT counts how many logic-1 bits exist in 8 bits of data.
2. This function is useful in performing parity check.

4.1.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input 8-bit data</td>
<td>R0L</td>
<td>1</td>
</tr>
<tr>
<td>Output Number of logic-1 bits</td>
<td>R1L</td>
<td>1</td>
</tr>
</tbody>
</table>

4.1.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0H</th>
<th>R0L</th>
<th>R1H</th>
<th>R1L</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

✓ : Unchanged
• : Indeterminate
† : Result
4.1.4 Specifications

<table>
<thead>
<tr>
<th>Program memory (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
</tr>
<tr>
<td>162</td>
</tr>
<tr>
<td>Reentrant</td>
</tr>
<tr>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
</tr>
<tr>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
</tr>
<tr>
<td>Possible</td>
</tr>
</tbody>
</table>

4.1.5 Notes

The specified clock cycle count (162) is for 8-bit data = "FF".

4.1.6 Description

1. Details of functions
   a. The following arguments are used with the software HCNT:
      R0L: Contains, as an input argument, 8-bit data that may have logic-1 bits to be counted.
      R1: Contains, as an output argument, the number of logic-1 bits that have been found and counted in the 8-bit data.
   b. Figure 4.1 shows an example of the software HCNT being executed. When the input argument is set as shown in (1), the number of logic-1 bits that have been found in the 8-bit data is placed in R1L as shown in (2).
   c. The contents of R0L are retained after execution of the software HCNT.
(1) Input arguments

- R0L
  - (8-bit data H'67)
  - 0 1 1 0 0 1 1 1
  - 5 logic-1 bits

(2) Output arguments

- R1L
  - (Logic-1 bit count H'05)
  - 0 5

Figure 4.1 Example of Software HCNT Execution

2. Notes on usage
   To count the logic-0 bits, complement the logic 1 in R0L (by using the NOT instruction) before executing the software HCNT.

3. Data memory
   The software HCNT does not use the data memory.

4. Example of use
   Set 8-bit data in the input argument and call the software HCNT as a subroutine.

   ```
   WORK1 .RES.B 1
   ...
   MOV. B @WORK1, R0L
   JSR           @HCNT
   ...
   RES. B     1
   ...
   ```

   - Reserves a data memory area in which the user program places 8-bit data.
   - Places in the input argument the 8-bit data set by the user program.
   - Calls the software HCNT as a subroutine.

5. Operation
   a. R1H is used as the counter that indicates an 8-bit data rotation count.
   b. The ROTXL instruction is used to set data (R0L) bit by bit in the C flag.
   c. R1L is incremented when the C flag is 1. No operation occurs when the C flag is 0.
   d. R1H is decremented each time the b.-c. process is performed. The process is repeated until R1H reaches 0.
4.1.7 Flowchart

HCNT

#H'0900 → R1

R1H - #1 → R1H

R1H = 0

YES

NO

YES

C flag is "0"

NO

R1L + #1 → R1L

EXIT

RTS

Places "9" in the counter (R1H) and zero-clear the counter (R1L) that counts logic-1 bits.

Decrement R1H until it reaches H'00.

Places the most significant bit of the 8-bit data in the C flag. Branches without counting up if the C flag is "0".

Increments R1L.
4.1.8 Program List

** Program Name =

```assembly
;********************************************************************
;* 00 - NAME :HIGH LEVEL BIT COUNT (HCNT)
;********************************************************************

; ENTRY :R0L (8 BIT DATA)

HCNT.cod C 0000 79010900 MOV.W #H'0900,R1
HCNT.cod C 0004 1A01 DEC R1H
HCNT.cod C 0006 4708 BCC LBL ;Branch if C flag = 0
HCNT.cod C 0008 1288 INC R1L ;Branch always
HCNT.cod C 000A 40F4 EXIT

;********************************************************************
```

*****TOTAL ERRORS 0
*****TOTAL WARNINGS 0
4.2 Shift 16-Bit Binary to Right (SHR)

MCU: H8/300 Series
     H8/300L Series

Label name: SHR

4.2.1 Function

1. The software SHR shifts a 16-bit binary number to the right.
2. Shift count: 1 to 16.
3. This function is useful in multiplying a 16-bit binary number by 2^n (n=shift count).

4.2.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit binary to be shifted right</td>
<td>R0</td>
<td>2</td>
</tr>
<tr>
<td>Shift count</td>
<td>R1L</td>
<td>1</td>
</tr>
</tbody>
</table>

| Output: Result of shift  | R0          | 2                   |

4.2.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1H</th>
<th>R1L</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>†</td>
<td>•</td>
<td>✗</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

=max: Unchanged
•: Indeterminate
†: Result
### 4.2.4 Specifications

<table>
<thead>
<tr>
<th>Program memory (bytes)</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>168</td>
</tr>
</tbody>
</table>

#### Notes

The specified clock cycle count (162) is for shifting right 16 bits.

#### Description

1. Details of functions
   a. The following arguments are used with the software SHR:
      - R0: Contains, as an input argument, a 16-bit binary number to be right-shifted. The result of shift is placed in R0 after execution of the software SHR.
      - R1L: Contains, as an input argument, the right-shift count of a 16-bit binary number.
   b. Figure 4.2 shows an example of the software SHR being executed. When the arguments are set as shown in (1), the 16-bit binary number is shifted right as shown in (2). 0's are placed in the remaining upper bits.
Figure 4.2  Example of Software SHR Execution

2. Notes on usage
   R1L must satisfy the condition H'01 ≤ R1L ≤ H'0F; otherwise, R0 contains all 0's.

3. Data memory
   The software SHR does not use the data memory.

4. Example of use
   Set a 16-bit binary number and a shift count in the input arguments and call the software SHR as a subroutine.

<table>
<thead>
<tr>
<th>WORK1</th>
<th>.RES. W</th>
<th>1</th>
<th>Reserves a data memory area in which the user program places the 16-bit binary number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORK2</td>
<td>.RES. B</td>
<td>1</td>
<td>Reserves a data memory area in which the user program places the 16-bit binary number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOV. W</td>
<td>@WORK1, R0</td>
<td>Places the 16-bit binary number set by the user program in the input argument.</td>
</tr>
<tr>
<td></td>
<td>MOV. B</td>
<td>@WORK2, R1L</td>
<td>Places the shift count set by the user program in the input argument.</td>
</tr>
<tr>
<td></td>
<td>JSR</td>
<td>@SHR</td>
<td>Calls the software SHR as a subroutine.</td>
</tr>
</tbody>
</table>

---

(1) Input arguments

\[
\begin{align*}
\text{R0}(\text{H'B3AD}) & \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \\
\text{R1L}(\text{H'02}) & \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1
\end{align*}
\]

(2) Output arguments

\[
\begin{align*}
\text{R0}(\text{H'2CEB}) & \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1
\end{align*}
\]
5. Operation
   a. The upper 8 bits of a 16-bit binary number is shifted right and the least significant bit in the C flag. Then the lower 8 bits are rotated right. This causes the least significant bit (in the C flag) moves to the most significant bit of the lower 8 bits.

   b. R1L is used as the counter that indicates the shift count. R1L is decremented each time the process a. is executed. This process is repeated until R1L reaches 0.
4.2.7 Flowchart

- **SHR**
  - Shift R0H 1 bit right
  - Rotate R0L 1 bit right
  - R1L -#1 → R1L

- **YES**
  - R1L ≠ 0

- **NO**
  - RTS

- **Branches if it is not 0.**

- **Shifts the 16-bit binary number 1 bit to the right.**

- **Decrement the shift counter (R1L) and branches if it is not 0.**
4.2.8 Program List

*** 88/100 ASSEMBLER VER 1.0B ** 08/18/92 09:51:29

PROGRAM NAME =

;******************************************************************************
1 00 - NAME : SHIFT OF 16 BIT DATA (SHR)

;******************************************************************************
2 ENTRY: R0 (16 BIT BINARY DATA)
3 R1L (SHIFT COUNTER)

;******************************************************************************
4 RETURN: R0 (16 BIT BINARY DATA)

;******************************************************************************
5

;SECTION SHR_code, CODE, ALIGN=2
14 SHR_code C 0000
15 .EXPORT SHR
16
17 SHR_code C 00000000 SHR .EQU $ ; Entry point
18 SHR_code C 0000 1100 SHLR R0H ; Shift 16 bit binary 1 bit right
19 SHR_code C 0002 1308 RTER R1L ; Decrement Shift counter
20 SHR_code C 0004 1A99 DEC R1L
21 SHR_code C 0006 44FB BRNE SHR ; Branch if not R1L=0
22 SHR_code C 0008 5470 RTS
23
24 .END

*****TOTAL ERRORS 0
*****TOTAL WARNINGS 0
Section 5  COUNTER

5.1  4-Digit Decimal Counter

MCU:  H8/300 Series  
       H8/300L Series

Label name:  DECNT

5.1.1  Function

1. The software DCNT increments a 4-digit binary-coded decimal (BCD) counter by 1.
2. This function is useful in counting interrupts (external interrupts, timer interrupts, etc.).

5.1.2  Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Output</td>
<td>4-digit BCD counter</td>
<td>DCNTR (RAM)</td>
</tr>
<tr>
<td></td>
<td>Counter overflow</td>
<td>C flag (CCR)</td>
</tr>
</tbody>
</table>

5.1.3  Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>×</td>
<td></td>
<td>×</td>
<td></td>
<td>×</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

× : Unchanged
• : Indeterminate
† : Result
5.1.4 Specifications

<table>
<thead>
<tr>
<th>Program memory (bytes)</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data memory (bytes)</td>
<td>2</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>28</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Impossible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

5.1.5 Description

1. Details of functions
   a. The following arguments are used with the software DECNT:
      - DCNT: Used as a 4-digit BCD counter that is incremented by 1 each time the software DECNT is executed.
      - C flag (CCR): Indicates the state of the counter after execution of the software DECNT.
        C flag = 1: The counter overflowed. (See figure 5.2.)
        C flag = 0: The counter was incremented normally.
   b. Figure 5.1 shows an example of the software DECNT being executed. When the software DECNT is executed, the 4-digit BCD counter is incremented as shown in (2).
2. Notes on usage

Figure 5.2 Example of Software DECNT Execution

<table>
<thead>
<tr>
<th>(1) Before execution of software</th>
<th>DCNTR(H'4099)</th>
<th>4</th>
<th>0</th>
<th>9</th>
<th>9</th>
</tr>
</thead>
</table>

(2) Output arguments

<table>
<thead>
<tr>
<th>DCNTR(H'4100)</th>
<th>4</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C flag</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.1 Example of Software DECNT Execution

<table>
<thead>
<tr>
<th>(1) Before execution of software</th>
<th>DCNTR(H'9999)</th>
<th>9</th>
<th>9</th>
<th>9</th>
<th>9</th>
</tr>
</thead>
</table>

(2) Output arguments

<table>
<thead>
<tr>
<th>DCNTR(H'0000)</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C flag</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.2 Example of Software DECNT Execution

3. Data memory

<table>
<thead>
<tr>
<th>Label name</th>
<th>Bit 7</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCNTR</td>
<td>Upper</td>
<td>Lower</td>
</tr>
</tbody>
</table>

Contains a 4-digit BCD counter value.
4. Example of use

<table>
<thead>
<tr>
<th>DCNTR</th>
<th>RES. W</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSH</td>
<td>R0</td>
<td></td>
</tr>
<tr>
<td>JSR</td>
<td>@DECNT</td>
<td></td>
</tr>
<tr>
<td>POP</td>
<td>R0</td>
<td></td>
</tr>
<tr>
<td>BCS</td>
<td>OVER</td>
<td></td>
</tr>
<tr>
<td>OVER</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|        |        | Reserves a data memory area that contains the count value of the 4-digit BCD counter. |
|--------|--------| Saves the register that is destroyed by the execution of software DECNT. |
|        |        | Calls the software DECNT as a subroutine. |
|        |        | Returns the register. |
|        |        | Branches to counter overflow processing routine when the BCD counter overflows. |

OVER Counter overflow processing routine

5. Operation
   a. The software DECNT uses data memory (DCNTR) as a 4-digit BCD counter.
   b. Each time the software DECNT is executed as a subroutine, DCNTR is incremented to repeat decimal correction.
5.1.6 Flowchart

```
DECNT

@DCNTR → R0

R0L + #1 → R0L

Decimal correction of R0L

R0H + #0 + C → R0H

Decimal correction of R0H

R0 → @DCNTR

RTS
```

- Places the 4-digit BCD counter (DCNTR) in R0.
- Performs decimal correction by incrementing R0L.
- Performs decimal correction by adding carry digits that occurred in R0H and R0L.
- Places the result in DCNTR.
5.1.7 Program List

*** H8/300 ASSEMBLER                VER 1.0B **   08/18/92 09:52:01
PROGRAM NAME =

1 ;********************************************************************
2 ;* 00-NAME :4 FIGURE BCD COUNTER(DECNT)
3 ;********************************************************************
4 ; ENTRY :NOTHING
5 ; RETURNS :DCNTR (BCD COUNTER)
6 ; C flag OF CCR (C=0 IS TRUE/C=1 IS OVER FLOW)
7 ;********************************************************************
8 ;
9 ;********************************************************************
10 ;
11 ;
12 ;********************************************************************
13 ;
14 DECNT_co C 0000
15 .SECTION DECNT_code,CODE,ALIGN=2
16 .EXPORT DECNT
17 ;
18 DECNT_co C 0000 68000000
19 ADD.W #H'01,R0L ;R0L + #H'01 -> R0L
20 DECNT_co C 0006 0F00
21 ADD X.B #H'00,R0H ;R0H + #H'00 + C -> R0H
22 DECNT_co C 000C 6B800000
23 MOV.W R0,@DCNTR ;Store R0 to DCNTR
24 DECNT_co C 0010 5470
25 RTS
26 ;
27 ;
28 DATA_dat D 0000
29 .SECTION DATA_data,DATA,ALIGN=2
30 .EXPORT DCNTR
31 DATA_dat D 0000 0000
32 ;
33 ;
34 ;
35 .END

*****TOTAL ERRORS       0
*****TOTAL WARNINGS     0

76
Section 6  COMPARISON

6.1 Compare 32-Bit Binary Numbers

MCU:  H8/300 Series
       H8/300L Series

Label name:  COMP

6.1.1 Function

1. The software COMP compares two 32-bit binary numbers and sets the result (> , =, <) in the C
   and Z flags (CCR).
2. All arguments are unsigned integers.

6.1.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Comparand</td>
<td>R0, R1</td>
</tr>
<tr>
<td>Source number</td>
<td>R2, R3</td>
<td>4</td>
</tr>
<tr>
<td>Output</td>
<td>Result of comparison</td>
<td>C flag, Z flag (CCR)</td>
</tr>
</tbody>
</table>

6.1.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I  U  H  U  N  Z  V  C

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

× : Unchanged
• : Indeterminate
‡ : Result
6.1.4 Specifications

<table>
<thead>
<tr>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>Reentrant</td>
</tr>
<tr>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
</tr>
<tr>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
</tr>
<tr>
<td>Possible</td>
</tr>
</tbody>
</table>

6.1.5 Description

1. Details of functions
   a. The following arguments are used with the software COMP:
      R0, R1: Contain a 32-bit binary comparand as input arguments. (See figure 6.1.)
      R2, R3: Contain a source 32-bit binary number as input arguments. (See figure 6.1.)

   ![Figure 6.1 Input Argument Setting](image-url)
b. Table 6.1 shows an example of the software COMP being executed.
The C and Z flags are set according to the input arguments.

<table>
<thead>
<tr>
<th>Input arguments</th>
<th>Large</th>
<th>Sign</th>
<th>Output arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R0</td>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>F67D</td>
<td>2001</td>
<td>&lt;</td>
<td>2200</td>
</tr>
<tr>
<td>2010</td>
<td>2020</td>
<td>=</td>
<td>2010</td>
</tr>
<tr>
<td>4001</td>
<td>F000</td>
<td>&gt;</td>
<td>A000</td>
</tr>
</tbody>
</table>

The input arguments are stored even after execution of the software COMP.

2. Notes on usage
   When not using upper bits, set 0's in them; otherwise, no correct result of comparison can be obtained because comparison is made on the numbers including indeterminate data set in the upper bits.

3. Data memory
   The software COMP does not use the data memory.
4. Example of use

Set a source binary number and a comparand in the input arguments and call the software COMP as a subroutine.

<table>
<thead>
<tr>
<th>WORK1</th>
<th>.RES. W 2</th>
<th>Reserves a data memory area in which the user program places a 32-bit binary comparand.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORK2</td>
<td>.RES. W 2</td>
<td>Reserves a data memory area in which the user program places a source 32-bit binary number.</td>
</tr>
</tbody>
</table>

```
MOV.W @WORK1, R0
MOV.W @WORK1+2, R1
MOV.W @WORK2, R2
MOV.W @WORK2+2, R3
```

Places the 32-bit binary comparand set by the user program.

Places the source 32-bit binary number set by the user program.

```
JSR @COMP
```

Calls the software COMP as a subroutine.

```
BEQ SKIP1
BCS SKIP2
```

Branches to the processing routine for comparand = source number.

Branches to the processing routine for comparand < source number.

```
BRA SKIP3
BRA SKIP3
```

Processing routine for comparand > source number

```
SKIP1
```

Processing routine for comparand = source number

```
SKIP2
```

Processing routine for comparand < source number

```
SKIP3
```

User program

5. Operation

a. Comparison of two or more words can be done by performing a series of 1-word comparisons.

b. The output arguments are the C and Z flags after execution of the compare instruction (CMP.W).

c. The upper words are compared by using the word compare instruction (CMP.W). If the upper words are not equal, the software COMP terminates. If the upper words are equal, then the lower words are compared.
### 6.1.6 Flowchart

```
COMP

NO
R0 = R2

YES
R1 - R3

LBL

RTS

---
<table>
<thead>
<tr>
<th>Compares the upper 16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compares the lower 16 bits</td>
</tr>
</tbody>
</table>
```

### 6.1.7 Program List

```assembly
*** H8/300 ASSEMBLER                VER 1.0B **  08/18/92 09:52:34
PROGRAM NAME =

1 ;********************************************************************
2 ;*
3 ;* 00 - NAME :32 BIT COMPARISON (COMP)
4 ;*
5 ;********************************************************************
6 ;*
7 ;* ENTRY :R0 (COMPARAND DATA HIGH)
8 ;* R1 (COMPARAND DATA LOW)
9 ;* R2 (COMPARATIVE DATA HIGH)
10 ;* R3 (COMPARATIVE DATA LOW)
11 ;*
12 ;* RETURNS :C flag & Z flag (COMPARISON RESULT)
13 ;*
14 ;********************************************************************
15 ;
16 COMP_code C 0000                .SECTION COMP_code,CODE,ALIGN=2
17                                      .EXPORT COMP
18 ;
19 COMP_code C 00000000              COMP_EQU $  ;Entry point
20 COMP_code C 0000 1D20             CMP.W R2,R0
21 COMP_code C 0002 4602             BNE LBL ;Branch if Z=0
22 COMP_code C 0004 1D31             CMP.W R3,R1
23 COMP_code C 0006 5470             RTS
24 COMP_code C 0006 5470             .END
25 ;
26 ;
*****TOTAL ERRORS       0
*****TOTAL WARNINGS     0
```
Section 7  ARITHMETIC OPERATION

7.1  Addition of 32-Bit Binary Numbers

MCU:  H8/300 Series
       H8/300L Series

Label name:  ADD1

7.1.1  Function

1. The software ADD1 adds a 32-bit binary number to another 32-bit binary number and places
   the result (a 32-bit binary number) in a general-purpose register.
2. The arguments used with the software ADD1 are unsigned integers.
3. All data is manipulated on general-purpose registers.

7.1.2  Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Augend</td>
<td>R0, R1</td>
<td>4</td>
</tr>
<tr>
<td>Addend</td>
<td>R2, R3</td>
<td>4</td>
</tr>
<tr>
<td>Output</td>
<td>Result of addition</td>
<td>RO, R1</td>
</tr>
<tr>
<td>Carry</td>
<td>C flag (CCR)</td>
<td></td>
</tr>
</tbody>
</table>

7.1.3  Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>†</td>
<td>†</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

I  U  H  U  N  Z  V  C
| •  | •  | ×   | •  | ×   | ×   | ×   | †  |

×  : Unchanged
•  : Indeterminate
†  : Result
7.1.4 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>8</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>14</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

7.1.5 Description

1. Details of functions
   a. The following arguments are used with the software ADD1:
      R0, R1: Contain a 32-bit binary augend as an input argument. The result of addition is placed in these registers after execution of the software ADD1.
      R2, R3: Contain a 32-bit binary addend as an input argument.
      C flag (CCR): Determines the presence or absence of a carry as an output argument after execution of the software ADD1.
      C flag = 1: A carry occurred in the result. (See figure 7.1)
      C flag = 0: No carry occurred in the result.
b. Figure 7.2 shows an example of the software ADD1 being executed. When the input arguments are set as shown in (1), the result of addition is placed in R0 and R1 as shown in (2).

![Figure 7.2 Example of Software ADD1 Execution](image-url)
2. Notes on usage

When upper bits are not used (see figure 7.3), set 0's in them; otherwise, no correct result can be obtained because addition is done on the numbers including indeterminate data.

![Figure 7.3  Example of Addition with Upper Bits Unused](image)

b. After execution of the software ADD1, the augend is destroyed because the result is placed in R0 and R1. If the augend is necessary after software ADD1 execution, save it on memory.

3. Data memory

The software ADD1 does not use the data memory.
4. Example of use
Set an augend and an addend in the input arguments and call the software ADD1 as a subroutine.

\[
\text{WORK1 .RES. W 2} \quad \text{Reserves a data memory area in which the user program places a 32-bit binary augend.}
\]

\[
\text{WORK2 .RES. W 2} \quad \text{Reserves a data memory area in which the user program places a 32-bit binary addend.}
\]

\[
\text{WORK3 .RES. W 2} \quad \text{Reserves a data memory area for storage of the result of addition.}
\]

\[
\begin{align*}
\text{MOV. W} & \quad \text{Places in the input arguments (R0 and R1) the 32-bit binary augend set by the user program.} \\
\text{MOV. W} & \quad \text{Places in the input arguments (R2 and R3) the 32-bit binary addend set by the user program.} \\
\text{MOV. W} & \quad \text{Calls the software ADD1 as a subroutine.} \\
\text{MOV. W} & \quad \text{Branches to the carry processing routine if a carry has occurred in the result.} \\
\text{MOV. W} & \quad \text{Places the result (set in the output arguments (R3 and R4)) in the data memory of the user program.}
\end{align*}
\]

\[
\text{JSR @ADD1} \quad \text{Calls the software ADD1 as a subroutine.}
\]

\[
\text{BCS OVER} \quad \text{Branches to the carry processing routine if a carry has occurred in the result.}
\]

\[
\text{MOV. W R0, @WORK3} \quad \text{Places the result (set in the output arguments (R3 and R4)) in the data memory of the user program.}
\]

\[
\text{MOV. W R1, @WORK3+2} \quad \text{Places the result (set in the output arguments (R3 and R4)) in the data memory of the user program.}
\]

\[
\text{OVER Carry processing routine}
\]

5. Operation
a. Addition of 3 bytes or more can be done by repeating 1-byte additions.
b. A 1-word add instruction (ADD.W), which does not involve the state of the C flag, is used to add the lower word shown by equation 1. The C flag is set if a carry occurs after execution of the equation.
\[
R1 + R3 \rightarrow R1 \quad \text{equation 1}
\]
c. A 1-byte add instruction (ADDX.B), which involves the state of the C flag, is used twice to add the upper word shown by equation 2.

\[
\begin{align*}
R0L + R2L + C & \rightarrow R0L \quad \text{equation 1} \\
R0H + R2H + C & \rightarrow R0H 
\end{align*}
\]

The C flag indicates a carry that may occur in the result of addition of the lower word executed in b. and the lower bytes of the upper word.

### 7.1.6 Flowchart

```
ADD1

R1 + R3 \rightarrow R1

\{ Adds the lower word and sets the result. \}

R0L + R2L + C \rightarrow R0L
R0H + R2H + C \rightarrow R0H

\{ Adds the upper word involving a carry (the state of the C flag) that may occur in the result of addition of the lower word, and sets the result. \}

RTS
```
7.1.7 Program List

**H8/300 ASSEMBLER**                VER 1.0B **   08/18/92 09:53:09**

**PROGRAM NAME =**

```
1 ;******************************************************************************
2 ;*                           00-NAME :32 BIT ADDITION (ADD1)
3 ;*                    931                 ADD.W R3,R1 ;Adjust lower word
4 ;*                              0040620 ;Adjust upper word
5 ;*                    5470                 RTS
6 ;*                   0931                 ADD.X.B R2L,R0L ;Adjust upper word
7 ;*                                21 ADD1 ;Entry point
8 ;*                   0000031 ;Adjust lower word
9 ;*                 .SECTION ADD1_code,CODE,ALIGN=2
10 ;*                               15 ADD1 .EQU $ ;Entry point
11 ;*                   0000000 ADD1 .EQU 5 ;Entry point
12 ;*                    0000 0931     ADD.W R3,R1 ;Adjust lower word
13 ;*                           ADD1 .EQU 5 ;Entry point
14 ;*                          ADD1 .EQU 5 ;Entry point
15 ;*                               .END
16 ;*'******************************************************************************
```

*****TOTAL ERRORS       0
*****TOTAL WARNINGS     0
7.2 Subtraction of 32-Bit Binary Numbers

MCU: H8/300 Series
H8/300L Series

Label name: SUB1

7.2.1 Function
1. The software SUB1 subtracts a 32-bit binary number from another 32-bit binary number and places the result (a 32-bit binary number) in a general-purpose register.
2. The arguments used with the software SUB1 are unsigned integers.
3. All data is manipulated on general-purpose registers.

7.2.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Minuend</td>
<td>R0, R1</td>
</tr>
<tr>
<td></td>
<td>Subtrahend</td>
<td>R2, R3</td>
</tr>
<tr>
<td>Output</td>
<td>Result of subtraction</td>
<td>R0, R1</td>
</tr>
<tr>
<td></td>
<td>Borrow</td>
<td>C flag (CCR)</td>
</tr>
</tbody>
</table>

7.2.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>†</td>
<td>†</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>†</td>
</tr>
</tbody>
</table>

† : Unchanged
× : Unchanged
• : Indeterminate
† : Result

I U H U N Z V C
• • × • × × × †
7.2.4 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>8</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>14</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

7.2.5 Description

1. Details of functions
   a. The following arguments are used with the software SUB1:
      R0, R1: Contain a 32-bit binary minuend as an input argument. The result of subtraction is placed in these registers after execution of the software SUB1.
      R2, R3: Contain a 32-bit binary subtrahend as an input argument.
      C flag (CCR): Determines the presence or absence of a borrow as an output argument after execution of the software SUB1.
      C flag = 1: A borrow occurred in the result. (See figure 7.4)
      C flag = 0: No borrow occurred in the result.
b. Figure 7.5 shows an example of the software SUB1 being executed. When the input arguments are set as shown in (1), the result of subtraction is placed in R0 and R1 as shown in (2).

![Diagram of subtraction with borrow]

**Figure 7.4** Example of Subtraction with a Borrow

**Figure 7.5** Example of Software SUB1 Execution
2. Notes on usage
   a. When upper bits are not used (see figure 7.6), set 0's in them; otherwise, no correct result can be obtained because subtraction is done on the numbers including indeterminate data.

   b. After execution of the software SUB1, the minuend is destroyed because the result is contained in R0 and R1. If the minuend is necessary after software SUB1 execution, save it on memory.

3. Data memory
   The software SUB1 does not use the data memory.
4. Example of use

Set a minuend and a subtrahend in the input arguments and call the software SUB1 as a subroutine.

```
WORK1 .RES. W 2
M       
M       
MOV. W @WORK1, R0
MOV. W @WORK1+2, R1
MOV. W @WORK2, R2
MOV. W @WORK2+2, R3
JSR @SUB1

WORK2 .RES. W 2
M       
M       
MOV. W @WORK1, R0
MOV. W @WORK1+2, R1

WORK3 .RES. W 2
M       
M       
MOV. W @WORK1, R0
MOV. W @WORK1+2, R1

MOV. W @WORK3, R0
MOV. W @WORK3+2, R1

BORROW  Borrow processing routine
```

5. Operation

a. Subtraction of 3 bytes or more can be done by repeating 1-byte subtractions.

b. A 1-word subtract instruction (SUB.W), which does not involve the state of the C flag, is used to subtract the lower word shown by equation 1. The C flag is set if a borrow occurs after execution of the equation.

\[
R1 - R3 \rightarrow R1 \quad \text{equation 1}
\]
c. A 1-byte subtract instruction (SUBX.B), which involves the state of the C flag, is used twice to subtract the upper word shown by equation 2.

\[
\begin{align*}
R0L - R2L - C & \rightarrow R0 \\
R0H - R2H - C & \rightarrow R0
\end{align*}
\]

\text{equation 2}

The C flag indicates a borrow that may occur in the result of subtraction of the lower word executed in b. and the lower bytes of the upper word.

7.2.6 Flowchart

\[
\begin{align*}
\text{SUB1} & \\
\text{R1 - R3} & \rightarrow \text{R1} & \text{Subtracts the lower word and sets the result.} \\
\text{R0L - R2L - C} & \rightarrow \text{R0L} \\
\text{R0H - R2H - C} & \rightarrow \text{R0H} & \text{Subtracts the upper word involving a borrow (the state of the C flag) that may occur in the result of subtraction of the lower word, and sets the result.} \\
\text{RTS} & 
\end{align*}
\]
7.2.7 Program List

*** HS/300 ASSEMBLER                VER 1.0B **   08/18/92 09:53:37

PROGRAM NAME =

---------------------------------------------------------------------
| 1 | ;******************************************************************** |
| 2 | ;* 00 - NAME : 32 BIT BINARY SUBTRACTION (SUB1)                     |
| 3 | ;* ;******************************************************************** |
| 4 | ;* ENTRY: R0,R1 (MINUEND)                                         |
| 5 | ;* R2,R3 (SUBTRAHEND)                                            |
| 6 | ;* RETURNS: R0,R1 (RESULT)                                       |
| 7 | ;* C flag of CCR (C=0; TRUE, C=1; BORROW)                        |
| 8 | ;******************************************************************** |

; SUB1_code C 0000                    .SECTION SUB1_code,CODE,ALIGN=2
11 SUB1_code C 0000 1931             SUB.W R3,R1 ; Subtract lower word
12 SUB1_code C 0002 1EA8             SUBX.B R2L,R0L ; Subtract upper word
13 SUB1_code C 0004 1E20             SUBX.B R2H,R0H
14 ;
15 SUB1_code C 0006 5470             RTS
16 ;
17 ;
18 SUB1_code C 00000000              SUB1 .EQU $ ; Entry point
19 SUB1_code C 0000 1931             SUB.W R3,R1 ; Subtract lower word
20 SUB1_code C 0002 1EE8             SUBX.B R2L,R0L ; Subtract upper word
21 SUB1_code C 0004 1E20             SUBX.B R2H,R0H
22 ;
23 SUB1_code C 0006 5470             RTS
24 ;
25 ;
****TOTAL ERRORS 0
****TOTAL WARNINGS 0
7.3 Multiplication of 16-Bit Binary Numbers

MCU:  H8/300 Series
       H8/300L Series

Label name:  MUL

7.3.1 Function

1. The software MUL multiplies a 16-bit binary number by another 16-bit binary number and places the result (a 32-bit binary number) in a general-purpose register.
2. The arguments used with the software MUL are unsigned integers.
3. All data is manipulated on general-purpose registers.

7.3.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplicand</td>
<td>R1</td>
<td>2</td>
</tr>
<tr>
<td>Multiplier</td>
<td>R0</td>
<td>2</td>
</tr>
<tr>
<td>Output</td>
<td>Result of multiplication</td>
<td>R1, R2</td>
</tr>
</tbody>
</table>

7.3.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>†</td>
<td>†</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

× : Unchanged
• : Indeterminate
† : Result
7.3.4 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>32</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>86</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

7.3.5 Description

1. Details of functions
   a. The following arguments are used with the software MUL:
      R0: Contains a 16-bit binary multiplier as an input argument.
      R1: Contains a 16-bit binary multiplicand as an input argument. The upper 2 bytes of the result are placed in this register after execution of the software MUL.
      R2: Contains the lower 2 bytes of the result as an output argument.

```
R0  16-bit binary multiplier
R1  16-bit binary multiplicand
```

Figure 7.7 Input Argument Setting
b. Figure 7.8 shows an example of the software MUL being executed. When the input arguments are set as shown in (1), the result of multiplication is placed in R1 and R2 as shown in (2).

\[
\begin{array}{c}
\text{R1 (H'0A0B6)} & \begin{array}{c}
A\quad 0\quad B\quad 6
\end{array} & \rightarrow \text{Multiplicand} \\
\text{R0 (H'1F6A)} & \begin{array}{c}
1\quad F\quad 6\quad A
\end{array} & \rightarrow \text{Multiplier}
\end{array}
\]

\[
\begin{array}{c}
\text{R1, R2} \\
(\text{H'13B955C}) & \begin{array}{c}
1\quad 3\quad B\quad 8\quad 9\quad 5\quad 5\quad C
\end{array} & \rightarrow \text{Result of multiplication}
\end{array}
\]

\textbf{Figure 7.8}  \textbf{Example of Software MUL Execution}

c. Table 7.1 lists the results of multiplication with 0's placed in the input arguments.

\textbf{Table 7.1}  \textbf{Results of Multiplication with 0's Placed in Input Arguments}

<table>
<thead>
<tr>
<th>Input argument</th>
<th>Output argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplicand (R1)</td>
<td>Multiplier (R0)</td>
</tr>
<tr>
<td>H'****</td>
<td>H'0000</td>
</tr>
<tr>
<td>H'0000</td>
<td>H'****</td>
</tr>
<tr>
<td>H'0000</td>
<td>H'0000</td>
</tr>
</tbody>
</table>

\textbf{Note:} H'**** is a hexadecimal number.

2. Notes on usage

a. When upper bits are not used (see figure 7.9), set 0's in them; otherwise, no correct result can be obtained because multiplication is made on the numbers including indeterminate data.

\[
\begin{array}{c}
\text{R1} \\
\begin{array}{c}
0\quad 0\quad F\quad A
\end{array} & \rightarrow \text{Multiplicand}
\end{array}
\]

\[
\begin{array}{c}
\text{R0} \\
\begin{array}{c}
0\quad 0\quad 5\quad B
\end{array} & \rightarrow \text{Multiplier}
\end{array}
\]

\[
\begin{array}{c}
\text{R1, R2} \\
\begin{array}{c}
0\quad 0\quad 0\quad 0\quad 5\quad 8\quad D\quad E
\end{array} & \rightarrow \text{Result of multiplication}
\end{array}
\]

\textbf{Figure 7.9}  \textbf{Example of Multiplication with Upper Bits Unused}

b. After execution of the software MUL, the multiplicand is destroyed because the result is placed in R1. If the multiplicand is necessary after software MUL execution, save it on memory.
3. Data memory

The software MUL does not use the data memory.

4. Example of use

Set a multiplicand and a multiplier in the input arguments and call the software MUL as a subroutine.

| WORK1   | .RES. W 1 | Reserves a data memory area in which the user program places a 16-bit binary multiplicand. |
| WORK2   | .RES. W 1 | Reserves a data memory area in which the user program places a 16-bit binary multiplier. |
| WORK3   | .RES. W 2 | Reserves a data memory area for storage of the result of multiplication (a 32-bit binary number). |

```
MOV. W @WORK1, R1  \( \text{Places in the input arguments (R1) the 16-bit binary multiplicand set by the user program.} \)
MOV. W @WORK2, R0  \( \text{Places in the input arguments (R0) the 16-bit binary multiplier set by the user program.} \)
```

```
JSR @MUL  \( \text{Calls the software MUL as a subroutine.} \)
```

```
MOV. W R1, @WORK3
MOV. W R2, @WORK3+2
```
(Places the result (the 32-bit binary number, set in the output arguments) in the data memory of the user program.)
5. Operation
   a. Figure 7.10 shows an example of multiplying 16-bit binary numbers.

   ![Diagram of software MUL execution](image)

   **Figure 7.10  Example of Software MUL Execution**

   Multiplication of 16-bit binary numbers consists of two stages as shown in figure 7.10: (3) finding two partial products with the MULXU instruction and adding the two results (6).

   b. The program runs in the following steps:
      
      (i) The MULXU instruction is used to obtain the result of the multiplicand (lower bytes) × the multiplier (lower bytes) ((1) in figure 7.10) and the result of the multiplicand (upper bytes) × the multiplier (lower bytes) ((2) in figure 7.10). Then these two results are added to obtain a partial product, that is, the multiplicand x the multiplier (lower bytes) ((3) in figure 7.10).
      
      (ii) The MULXU instruction is used to obtain another partial product, that is, the multiplicand x the multiplier (upper bytes) ((6) in figure 7.10).
(iii) The two partial products (i) and (ii) are added to obtain the final product ((4) in figure 7.10).

### 7.3.6 Flowchart

![Flowchart Diagram]

- **MUL**
  - \( R1L \rightarrow R2L \)
  - \( R1H \rightarrow R4L \)
  - \( R1L \rightarrow R3L \)
  - \( R1H \rightarrow R1L \)

- Multiplicand (lower bytes) \( \times \) multiplier (lower bytes)
- Multiplicand (upper bytes) \( \times \) multiplier (lower bytes)
- Multiplicand (lower bytes) \( \times \) multiplier (upper bytes)
- Multiplicand (upper bytes) \( \times \) multiplier (upper bytes)

- To perform the MULXU instruction, a multiplicand (in bytes) in the lower bytes of the general-purpose register is set.

- \( R2 \times R0L \rightarrow R2 \)
- \( R4 \times R0L \rightarrow R4 \)
- \( R3 \times R0H \rightarrow R3 \)
- \( R1 \times R0H \rightarrow R1 \)

- Obtains partial product of multiplicand \( \times \) multiplier (lower bytes).

- \( R2H + R4L \rightarrow R2H \)
- \( R4H + \#0 + C \rightarrow R4H \)

- Obtains partial product of multiplicand \( \times \) multiplier (upper bytes).

- \( R1L + R3H \rightarrow R1L \)
- \( R1H + \#0 + C \rightarrow R1H \)

- Adds the partial products to obtain a final result.

- RTS
### Program List

**Program Name =**

```assembly
;******************************************************************************
;* 00 - NAME :16 BIT MULTIPLICATION (MUL)
;******************************************************************************

; ENTRY :R0 (MULTIPLIER)
; R1 (MULTIPLICAND)

; RETURNS :R1 (UPPER WORD OF RESULT)
; R2 (LOWER WORD OF RESULT)

;******************************************************************************
;
```

```assembly
MUL_code C 0000

.SECTION MUL_code,CODE,ALIGN=2

.EXPORT MUL

MUL_code C 0000 0000

MUL .EQU $ ;Entry point

MUL_code C 0000 0000 0C9A

MOV.B R1L,R2L ;R1L -> R2L

MUL_code C 0000 0000 0C1C

MOV.B R1H,R4L ;R1H -> R4L

MUL_code C 0000 0000 0C98

MOV.B R1L,R3L ;R1L -> R3L

MUL_code C 0000 0000 0C19

MOV.B R1H,R1L ;R1H -> R1L

MUL_code C 0000 0000 0C08 5082

MULXU R0L,R2 ;R0L * R2L -> R2

MUL_code C 0000 0000 0C08 5084

MULXU R0L,R4 ;R0L * R4L -> R4

MUL_code C 0000 0000 0C0C 5003

MULXU R0H,R3 ;R0H * R3L -> R3

MUL_code C 0000 0000 0C0E 5001

MULXU R0H,R1 ;R0H * R1L -> R1

MUL_code C 0000 0010 08C2

ADD.B R4L,R2H ;R2H + R4L + C -> R2H

MUL_code C 0012 0014 0839

ADD.B R3H,R1L ;R3H + R1L + C -> R1L

MUL_code C 0016 001E 0000

ADDX.B #H'00,R1H ;R1H + #H'00 + C -> R1H

MUL_code C 0018 0018 08B2

ADD.B R3L,R2H ;R3L + R2H + C -> R2H

MUL_code C 001A 001A 0849

ADDX.B #H'00,R1H ;R1H + #H'00 + C -> R1H

MUL_code C 001C 001E 0470

RTS

;******************************************************************************

*****TOTAL ERRORS       0

*****TOTAL WARNINGS     0

```

---

**Renesas**
7.4 Division of 32-Bit Binary Numbers

MCU:   H8/300 Series
       H8/300L Series

Label name:  DIV

7.4.1 Function

1. The software DIV divides a 32-bit binary number by another 32-bit binary number and places the result (a 32-bit binary number) in a general-purpose register.
2. The arguments used with the software DIV are unsigned integers.
3. All data is manipulated on general-purpose registers.

7.4.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dividend</td>
<td>R0, R1</td>
<td>4</td>
</tr>
<tr>
<td>Divisor</td>
<td>R2, R3</td>
<td>4</td>
</tr>
<tr>
<td>Output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result of division (Quotient)</td>
<td>R0, R1</td>
<td>4</td>
</tr>
<tr>
<td>Result of division (Remainder)</td>
<td>R4, R5</td>
<td>4</td>
</tr>
<tr>
<td>Errors</td>
<td>Z flag (CCR)</td>
<td></td>
</tr>
</tbody>
</table>

7.4.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>†</td>
<td>†</td>
<td>×</td>
<td>×</td>
<td>†</td>
<td>†</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>×</td>
<td>*</td>
<td>×</td>
<td>×</td>
<td>†</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

×: Unchanged
*: Indeterminate
†: Result
7.4.4 Specifications

<table>
<thead>
<tr>
<th>Program memory (bytes)</th>
<th>58</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>1374</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

7.4.5 Description

1. Details of functions
   a. The following arguments are used with the software DIV:
      R0: Contains the upper 2 bytes of a 32-bit binary dividend. The upper 2 bytes of the result of division (quotient) are placed in this register after execution of the software DIV.
      R1: Contains the lower 2 bytes of the 32-bit binary dividend. The lower 2 bytes of the result of division (quotient) are placed in this register after execution of the software DIV.
      R2: Contains the upper 2 bytes of a 32-bit binary divisor as an input argument.
      R3: Contains the lower 2 bytes of the 32-bit binary divisor as an input argument.
      R4: The upper 2 bytes of the result of division (remainder) are placed in this register as an output argument.
      R5: The lower 2 bytes of the result of division (remainder) are placed in this register as an output argument.
Z flag (CCR): Determines the presence or absence of an error (division by 0) with the software DIV as an output argument.

Z flag = 1: The divisor was 0.
Z flag = 0: The divisor was not 0.

**Figure 7.11  Input Argument Setting**

b. Figure 7.12 shows an example of the software DIV being executed. When the input arguments are set as shown in (1), the result of division is placed as shown in (2).

**Figure 7.12  Example of Software DIV Execution**

c. Table 7.2 lists the results of division with 0's placed in the input arguments.

**Table 7.2  Results of Division with 0's Placed in Input Arguments**

<table>
<thead>
<tr>
<th>Input argument</th>
<th>Output argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dividend (R0, R1)</td>
<td>Divisor (R2, R3)</td>
</tr>
<tr>
<td>H'********</td>
<td>H'00000000</td>
</tr>
<tr>
<td>H'00000000</td>
<td>H'********</td>
</tr>
<tr>
<td>H'00000000</td>
<td>H'00000000</td>
</tr>
</tbody>
</table>

Note: H'**** is a hexadecimal number.
2. Notes on usage
   When upper bits are not used (see figure 7.13), set 0's in them; otherwise, no correct result can be obtained because division is done on the numbers including indeterminate data.

   ![Figure 7.13 Example of Division with Upper Bits Unused](image)

   b. After execution of the software DIV, the dividend is destroyed because the quotient is placed in R0 and R1. If the dividend is necessary after software DIV execution, save it on memory.

3. Data memory
   The software DIV does not use the data memory.

4. Example of use
   Set a dividend and a divisor in the input arguments and call the software DIV as a subroutine.
WORK1 .RES. W 2

WORK2 .RES. W 2

WORK3 .RES. W 2

WORK4 .RES. W 2

MOV. W @WORK1, R0
MOV. W @WORK1+2, R1
MOV. W @WORK2, R2
MOV. W @WORK2+2, R3

JSR @DIV

BEQ ERROR

MOV. W R0, @WORK3
MOV. W R1, @WORK3+2
MOV. W R4, @WORK4
MOV. W R5, @WORK4+2

ERROR Error processing routine

Reserves a data memory area in which the user program places a 32-bit binary remainder.

Contains the 32-bit binary dividend set by the user program.

Contains the 32-bit binary divisor set by the user program.

Reserves a data memory area in which the user program places a 16-bit binary divisor.

Reserves a data memory area in which the user program places a 32-bit binary quotient.

Reserves a data memory area in which the user program places a 32-bit binary remainder.

Calls the software DIV as a subroutine.

Branches to the error processing routine if an error has occurred as the result of division.

Places the result of division (set in the output arguments) in the data memory of the user program.
5. Operation

a. A binary division can be done by performing a series of subtractions. Figure 7.14 shows an example of division (H'0D ÷ H'03).

![Division Diagram](image)

Figure 7.14 Example of Software DIV Execution (H'0D ÷ H'03)

This example indicates that the quotient and remainder are obtained by repeating a process of subtracting the dividend from the divisor. More specifically, the dividend is taken out bit by bit from the upper byte and the divisor is subtracted from the sum of the data and the previous result of subtraction.

b. The program runs in the following steps:
   (i) A shift count (D732) is set.
   (ii) The dividend is 1 bit shifted to the left to place the most significant bit in the least significant bit of the remainder.
   (iii) The divisor is subtracted from the remainder.
       If the result is positive, "1" is placed in the least significant bit of the dividend and the divisor is added to the result to return to the state before the subtraction. ((1) → (2) → (3) in figure 7.14). If the result is negative, "0" is placed in the least significant bit of the dividend and the divisor is added to the result to return to the state before the subtraction. ((4) → (5) → (6) in figure 7.14).
   (iv) The shift count (set in step (i)) is decremented.
   (v) Steps (ii) to (iv) are repeated until the shift count reaches H'00.
7.4.6 Flowchart

DIV

- #H'0000 → R4
  - R4 → R5

- R4 = R3
  - NO → YES

- R5 = R3
  - NO → YES

- LBL1

- #D'32 → R6L

- LBL2

- Shift R1 1 bit left
- Rotate R0 1 bit left
- Rotate R5 1 bit left
- Rotate R4 1 bit left

- #1 → Bit 0 of R1L

- R5 - R3 → R5
  - R4L - R2L + C → R4L
  - R4H - R2H + C → R4H

- C = 1
  - NO

- R5 + R3 → R5
  - R4L + R2L + C → R4L
  - R4H + R2H + C → R4H

- #0 → Bit 0 of R1L

- LBL3

- Clears R4 and R5 to 0.

- Exits if the divisor (R2, R3) is 0.

- Places a loop count (D'32) in the shift counter (R6L).

- Shifts the remainder (R4, R5) 1 bit to the left and places the most significant bit of the dividend (R0, R1) in the least significant bit of the remainder.

- Places 1 in the least significant bit of R1L.

- Subtracts the divisor (R2, R3) from the remainder (R4, R5).

- Branches if the result of subtraction is positive. If negative, adds the divisor (R2, R3) to the remainder (R4, R5) and places 0 in the least significant bit of R1L.
R6L - #1 → R6L

Decrements R6L (loop count).

R6 = 0

Branches if R6L is not 0.

0 → Z flag

Places 0 in the Z flag (CCR).

LBL2

EXIT

RTS
7.4.7 Program List

*** 80/300 ASSEMBLER               VER 1.0B **  08/18/92 09:58:57

PROGRAM NAME =

;********************************************************************
;* 00 - NAME : 32 BIT DIVISION (DIV)
;********************************************************************

ENTRY:R0 (UPPER WORD DIVIDEND)
*  R1 (LOWER WORD DIVIDEND)
R2 (UPPER WORD DIVISOR)
*  R3 (LOWER WORD DIVISOR)

* Z flag of CCR (Z=0;TRUE , Z=1;FALSE)

;********************************************************************

DIV_code C 0000 .SECTION DIV_code,CODE,ALIGN=2
DIV .EXPORT DIV

DIV_code C      00000000 .EQU $ ;Entry point
DIV_code C 0000 79040000 MOV.W #H'0000,R4 ;Clear R4
DIV_code C 0004 0D45 MOV.W R4,R5 ;Clear R5
DIV_code C 0006 1D42 CMP.W R4,R2
DIV_code C 0008 4604 BNE LBL1 ;Branch if Z flag = 0
DIV_code C 000A 1D43 CMP.W R4,R3
DIV_code C 000C 472A BEQ EXIT ;Branch if Z flag = 1 then exit

;  LBL1
DIV_code C 000E MOV.B #D'32,R6L ;Set byte counter
DIV_code C 0010 LBL2 ;
DIV_code C 0010 1009 SHLL R1L ;Shift dividend 1 bit left
DIV_code C 0012 1201 ROTEL R1H
DIV_code C 0014 1208 ROTEL R0L
DIV_code C 0016 1200 ROTEL R0H ;
DIV_code C 0018 7009 BSET #0,R1L ;Bit set bit 0 of R1L
DIV_code C 001A 1935 SUB.W R3,R5 ;R5 - R3 -> R5
DIV_code C 001C 1EAC SUBX.B R2L,R4L ;R4L - R2L - C -> R4L
DIV_code C 001E 0E24 SUBX.B R2H,R4H ;R2H - R4H + C -> R4H
DIV_code C 0020 7209 BCLR #0,R1L ;Bit clear bit 0 of R1L
DIV_code C 0022 1935 SUB.W R3,R5 ;R5 - R3 -> R5
DIV_code C 0024 1EAC SUBX.B R2L,R4L ;R4L - R2L - C -> R4L
DIV_code C 0026 1E24 SUBX.B R2H,R4H ;R2H - R4H + C -> R4H
DIV_code C 0028 4408 ;CCC LBL3 ;Branch if C=0
DIV_code C 002A 0935 ADD.W R3,R5 ;R3 + R5 -> R3
DIV_code C 002C 2214 ADDC.B R2L,R4L ;R2L + R4L + C -> R4L
DIV_code C 002E 2224 ADDC.B R2H,R4H ;R2H + R4H + C -> R4H
DIV_code C 0030 7209 BCLR #0,R4L ;Bit clear bit 0 of R4L
DIV_code C 0032 LBL3 ;
DIV_code C 0034 1A0E DEC.B R6L ;Decrement R6L
DIV_code C 0036 460A ;BEE LBL2 ;Branch if Z=0
DIV_code C 0038 06FB ;ABDC #B'1111011,CCR ;Clear Z flag
EXIT

;  RTS

*****TOTAL ERRORS       0
*****TOTAL WARNINGS     0

RENECHAS
7.5 Addition of Multiple-Precision Binary Numbers

MCU: H8/300 Series
     H8/300L Series

Label name: ADD2

7.5.1 Function

1. The software ADD2 adds a multiple-precision binary number to another multiple-precision binary number and places the result in the data memory where the augend was placed.
2. The arguments used with the software ADD2 are unsigned integers, each being up to 255 bytes long.

7.5.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Augend and addend byte length</td>
<td>R0L</td>
<td>1</td>
</tr>
<tr>
<td>Start address of augend</td>
<td>R3</td>
<td>2</td>
</tr>
<tr>
<td>Start address of addend</td>
<td>R4</td>
<td>2</td>
</tr>
<tr>
<td>Output Start address of the result of addition</td>
<td>R3</td>
<td>2</td>
</tr>
<tr>
<td>Error Z flag (CCR)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carry C flag (CCR)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7.5.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>†</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

I UH U U N Z V C

* : Unchanged
• : Indeterminate
† : Result
### 7.5.4 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>42</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>7170</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

### 7.5.5 Notes

The clock cycle count (7170) in the specifications is for addition of 255 bytes to 255 bytes.

### 7.5.6 Description

1. Details of functions
   a. The following arguments are used with the software ADD2:
      R0L: Contains, as an input argument, the byte count of an augend and an addend in 2-digit hexadecimal.
      R3: Contains the start address of the augend in the data memory area. The start address of the result of addition is placed in this register after execution of the software ADD2.
      R4: Contains, as an input argument, the start address of the addend in the data memory area.
      Z flag (CCR): Indicates an error in data length as an output argument.
Z flag = 0: The data byte count (R0L) was not 0.
Z flag = 1: The data byte count (R0L) was 0 (indicating an error).

C flag (CCR): Determines the presence or absence of a carry, as an output argument, after execution of the software ADD2.

C flag = 0: No carry occurred in the result of addition.
C flag = 1: A carry occurred in the result of addition (see figure 17-2).

b. Figure 7.15 shows an example of the software ADD2 being executed. When the input arguments are set as shown in (1), the result of addition is placed in the data memory area as shown in (2).

Figure 7.15   Example of Software ADD2 Execution

Figure 7.16 shows an example of addition with a carry that occurred in the result.

Figure 7.16   Example of Addition with a Carry
2. Notes on usage
   
a. When upper bits are not used (see figure 7.17), set 0's in them. The software ADD2 performs byte-based addition; if 0's are not set in the upper bits unused, no correct result can be obtained because the addition is done on the numbers including indeterminate data.

   ![Figure 7.17 Example of Addition with Upper Bits Unused](image)

b. After execution of the software ADD2, the augend is destroyed because the result is placed in the data memory area where the augend was set. If the augend is necessary after software ADD2 execution, save it on memory.

3. Data memory
   
The software ADD2 does not use the data memory.
4. Example of use

This is an example of adding 8 bytes of data. Set the start addresses of a byte count, an augend and an addend in the registers and call the software ADD2 as a subroutine.

```
WORK1 .RES.B 1
WORK2 .RES.B 8
WORK3 .RES.B 8

MOV. B @WORK1, R0L
MOV. W #WORK2, R3
MOV. W #WORK3, R4
JSR @ADD2

BCS OVER

OVER Carry processing routine.
```

Reserves a data memory area in which the user program places a byte count.
Reserves a data memory area in which the user program places an 8-byte binary augend.
Reserves a data memory area in which the user program places an 8-byte binary addend.
Places in the input argument (R0L) the byte count set by the user program.
Places in the input argument (R3) the start address of the augend set by the user program.
Places in the input argument (R4) the start address of the addend set by the user program.
Calls the software ADD2 as a subroutine.
Branches to the carry processing routine if a carry has occurred in the result of addition.

5. Operation

a. Addition of multiple-precision binary numbers can be done by performing a series of add instructions with a carry flag (ADDX.B) as the augend and addend data are placed in registers on a byte basis.

b. The end address of the data memory area containing the augend is placed in R3, and the end address of the data memory area containing the addend is placed in R4.

c. R1L is cleared for saving the C flag.

d. The augend and addend are loaded in R2L and R2H respectively, byte by byte, starting at their end address and equation 1 is executed:

\[
\text{Augend + addend + C} \rightarrow R2L \\
R2L \rightarrow @R3
\]

where the C flag indicates a carry that may occur in the result of addition of the lower bytes.
e. The result of d. is placed in the data memory area for the augend.

f. R3, R4, and R0L are decremented each time the process d. to e. terminates. This processing is repeated until R0L reaches 0.

7.5.7 Flowchart

```
ADD2

#H'00 → R0H
R0L → R1H

YES
R0L = 0
NO

R3 → R5

R0 - #1 → R0

R0 + R5 → R5
R0 + R4 → R4

#H'00 → R1L

①

② Exit if R0L reaches 0.

Sets R5 to the end address of the augend and the start address of the addend (R4) to the end address.

Clears R0H and copies the contents of R0H to the byte counter (R1H).

Copies the start address of the augend (R3) to R5.

Clears R1L (where the C flag is to be saved) to 0.
```
Adds the augend, addend and carry and places the result in the data memory area where the augend was placed.

Repeats this process as many times as the byte count of the addition data.

Sets bit 0 of R1L in the C flag.

Clears the Z flag to 0.

Note: ADD2 is the same as SUB2, ADDD2, and SUBD2 except for the step surrounded by dotted lines.
Program List

```
1 ;********************************************************************
2 ;* 00 - NAME : MULTIPLE PRECISION BINARY ADDITION
3 ;*
4 ;* 00 - NAME : MULTIPLE PRECISION BINARY ADDITION
5 ;*
6 ;********************************************************************
7 ;*
8 ; ENTRY : R0L (BYTE LENGTH OF ADDITION DATA)
9 ;* R3 (START ADDRESS OF SUMMANN)
10 ;* R4 (START ADDRESS OF ADDEND)
11 ;*
12 ;* RETURNS : R3 (START ADDRESS OF RESULT)
13 ;*  Z flag OF CCR (Z=0; TRUE , Z=1;FALSE)
14 ;* C flag OF CCR (C=0;TRUE , C=1;OVER FLOW)
15 ;*
16 ;********************************************************************
17 ;
18 ADD2_code C 0000 .SECTION ADD2_code,CODE,ALIGN=2
19 .EXPORT ADD2
20 ;
21 ADD2_code C 00000000 ADD2 .EQU $ ;Entry point
22 ADD2_code C 0000 F000 MOV.B #H'00,R0H ;Clear R0H
23 ADD2_code C 0002 0C81 MOV.B R0L,R1H ;Set byte counter(R1H)
24 ADD2_code C 0004 4722 BEQ EXIT ;Branch if R0L=0
25 ADD2_code C 0006 0D35 MOV.W R3,R5 ;R3 -> R5
26 ADD2_code C 0008 MAIN
27 ADD2_code C 0008 1B00 SUBS.W #1,R0 ;Decrement R0
28 ADD2_code C 000A 0905 ADD.W R0,R5 ;Set start address of summann(R5)
29 ADD2_code C 000C 0904 ADD.W R0,R4 ;Set start address of addend(R4)
30 ADD2_code C 000E F900 MOV.B #H'00,R1L ;Clear R1L
31 ADD2_code C 0010 LOOP
32 ADD2_code C 0010 685A MOV.B @R5,R2L ;Load summann to R2L
33 ADD2_code C 0012 6842 MOV.B @R4,R2H ;Load addend to R2H
34 ADD2_code C 0014 7709 BLD #0,R1L ;Load bit 0 of R1L to C flag
35 ADD2_code C 0016 0E2A ADDX.B R2H,R2L ;Addition
36 ADD2_code C 0018 6709 BST #0,R1L ;Store C flag to bit 0 of R1L
37 ADD2_code C 001A 680A MOV.B R2L,R5 ;Store result
38 ADD2_code C 001C 1805 SUBS.W #1,R5 ;Decrement summann address(R5)
39 ADD2_code C 001E 1804 SUBS.W #1,R4 ;Decrement addend address(R4)
40 ADD2_code C 0020 1A01 DEC.B R1H ;Decrement byte counter(R1H)
41 ADD2_code C 0022 46EC BNE LOOP ;Branch if Z=0
42 ;
43 ADD2_code C 0024 7709 BLD #0,R1L ;Load bit 0 of R1L to c flag
44 ADD2_code C 0026 06FB ANDC.B #H'FB,CCR ;Clear Z flag
45 ADD2_code C 0028 EXIT
46 ADD2_code C 0028 5470 RTS
47 ;
48 .END
```
7.6 Subtraction of Multiple-Precision Binary Numbers

MCU:  H8/300 Series
       H8/300L Series

Label name: SUB2

7.6.1 Function

1. The software SUB2 subtracts a multiple-precision binary number from another multiple-precision binary number and places the result in the data memory where the minuend was set.
2. The arguments used with the software SUB2 are unsigned integers, each being up to 255 bytes long.

7.6.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Minuend and subtrahend byte count</td>
<td>R0L</td>
<td>1</td>
</tr>
<tr>
<td>Start address of minuend</td>
<td>R3</td>
<td>2</td>
</tr>
<tr>
<td>Start address of subtrahend</td>
<td>R4</td>
<td>2</td>
</tr>
<tr>
<td>Output Start address of result</td>
<td>R3</td>
<td>2</td>
</tr>
<tr>
<td>Error Z flag (CCR)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Borrow C flag (CCR)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7.6.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>†</td>
<td>×</td>
<td>×</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>•</td>
<td>×</td>
<td>•</td>
<td>×</td>
<td>†</td>
<td>×</td>
<td>†</td>
</tr>
</tbody>
</table>

× : Unchanged
• : Indeterminate
† : Result
### 7.6.4 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>42</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>7170</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

### 7.6.5 Notes

The clock cycle count (7170) in the specifications for subtraction of 255 bytes from 255 bytes.

### 7.6.6 Description

1. Details of functions
   a. The following arguments are used with the software SUB2:
      - R0L: Contains, as an input argument, the byte count of a minuend and the byte count of a subtrahend in 2-digit hexadecimals.
      - R3: Contains, as an input argument, the start address of the data memory area where the minuend is placed. After execution of the software SUB2, the start address of the result is placed in this register.
      - R4: Contains, as an input argument, the start address of the data memory area where the subtrahend is placed.
      - Z flag (CCR): Indicates an error in data length as an output argument.
Z flag = 0: The data byte count (R0L) was not 0.
Z flag = 1: The data byte count (R0L) was 0, indicating an error.
C flag (CCR): Determines the presence or absence of a borrow after software SUB2 execution as an output argument.
C flag = 0: No borrow occurred in the result.
C flag = 1: A borrow occurred in the result. (See figure 7.19)
b. Figure 7.18 shows an example of the software SUB2 being executed. When the input arguments are set as shown in (1), the result of subtraction is placed in the data memory area as shown in (2).

![Table](image)

**Figure 7.18** Example of Software SUB2 Execution

Figure 7.19 shows an example of subtraction with a borrow that has occurred in the result.

![Diagram](image)

**Figure 7.19** Example of Subtraction with a Borrow
2. Notes on usage
   a. When upper bits are not used (see figure 7.20), set 0's in them. The software SUB2 performs byte-based subtraction; if 0's are not set in the upper bits unused, no correct result can be obtained because the subtraction is done on the numbers including indeterminate data.

   ![Figure 7.20 Example of Subtraction with Upper Bits Unused](image)

   b. After execution of the software SUB2, the minuend is destroyed because the result is placed in the data memory area where the minuend was set. If the minuend is necessary after software SUB2 execution, save it on memory.

3. Data memory
   The software SUB2 does not use the data memory.

4. Example of use
   This is an example of subtracting 8 bytes of data. Set the start addresses of a byte count, a minuend and a subtrahend in the registers and call the software SUB2 as a subroutine.
5. Operation

a. Subtraction of multiple-precision binary numbers can be done by repeating a subtract instruction with a carry flag (SUBX.B) as the minuend and subtrahend data are placed in registers on a byte basis.

b. The end address of the data memory area containing the minuend is placed in R3, and the end address of the data memory area containing the subtrahend is placed in R4.

c. R1L is cleared for saving the C flag.

d. The minuend and subtrahend are loaded in R2L and R2H respectively, byte by byte, starting at their end address and equation 1 is executed:

\[
\begin{align*}
\text{Minuend} - \text{subtrahend} - C & \rightarrow R2L \\
R2L & \rightarrow @R3
\end{align*}
\]

equation 1

where the C flag indicates a carry that may occur in the result of subtraction of the lower bytes.

e. The result of d. is placed in the data memory area for the minuend.

f. R3, R4, and R0L are decremented each time the process d. to e. terminates. This processing is repeated until R0L reaches 0.
7.6.7 Flowchart

- **SUB2**
  - \#H'00 \(\rightarrow\) \(R0H\)
  - \(R0L\) \(\rightarrow\) \(R1H\)
  - Stages:
    1. Clears \(R0H\) and copies the contents of \(R0H\) to the byte counter (\(R1H\)).
    2. Exits if \(R0L\) reaches 0.
    3. Copies the start address of the minuend (\(R3\)) to \(R5\).
    4. Sets \(R5\) to the end address of the minuend and the start address of the subtrahend (\(R4\)) to the end address.
    5. Clears \(R1L\) (where the C flag is to be saved) to 0.
LOOP

@R5 → R2L
@R4 → R2H

Bit 0 of R1L → C Flag

R2L + R2H + C → R2L

C Flag → Bit 0 of R1L

R2L → @R5

R5 - #1 → R5
R4 - #1 → R4

R1H - #1 → R1H

NO

R1H = 0

YES

Bit 0 of R1L → C Flag

0 → Z Flag

EXIT

RTS

Subtracts the subtrahend and borrow from the minuend and places the result in the data memory area where the minuend is placed.

Repeats this process as many times as the byte count of the subtraction data.

Sets bit 0 of R1L in the C flag.

Clears the Z flag to 0.

Note: SUB2 is the same as ADD2, ADDD2, and SUBD2 except for the step surrounded by dotted lines.
7.6.8 Program List

*** H8/300 ASSEMBLER    VER 1.0B **  08/18/92 10:00:06

PROGRAM NAME =

1                                      ;********************************************************************
2                                      ;* 00 - NAME :MULTIPLE-PRECISION BINARY SUBTRACTION
3                                      ;* 00 - NAME :MULTIPLE-PRECISION BINARY SUBTRACTION
4                                      ;* (SUB2)
5                                      ;*
6                                      ;********************************************************************
7                                      ;*
8                                      ;* ENTRY :ROL (BYTE LENGTH OF DATA)
9                                      ;* R3 (START ADDRESS OF MINUEND)
10                                     ;* R4 (START ADDRESS OF SUBTRAHEND)
11                                     ;*
12                                     ;* RETURNS :R3 (START ADDRESS OF RESULT)
13                                     ;* Z flag OF CCR (Z=0;TRUE , Z=1;FALSE)
14                                     ;* C flag OF CCR (C=0;TRUE , C=1;BORROW)
15                                     ;*
16                                     ;********************************************************************
17                                     ;
18 SUB2_cod C 0000                      .SECTION SUB2_code,CODE,ALIGN=2
19                                      .EXPORT SUB2
20                                      
21                                     ;
22 SUB2_cod C 0000 F000                  MOV.B #H'00,R0H ;Clear R0H
23 SUB2_cod C 0002 0C81                 MOV.B ROL,R1H ;Set byte counter(R1H)
24 SUB2_cod C 0004 4722                 BEQ EXIT ;Branch if R0L=0
25 SUB2_cod C 0006 0D35                 MOV.W R3,R3 ;R3 -> R5
26 SUB2_cod C 0008                      MAIN
27 SUB2_cod C 0008 1B00                 SUBS.W #1,R0 ;Decrement R0
28 SUB2_cod C 000A 0905                 ADD.W R0,R5 ;Adjust minuend start address(R5)
29 SUB2_cod C 000C 0904                 ADD.W R0,R4 ;Adjust subtrahend start address(R4)
30 SUB2_cod C 000E F900                 MOV.B #H'00,R1L ;Clear R1L
31 SUB2_cod C 0010                      LOOP
32 SUB2_cod C 0010 685A                 MOV.B @R5,R2L ;Load minuend
33 SUB2_cod C 0012 6842                 MOV.B @R4,R2H ;Load subtrahend
34 SUB2_cod C 0014 7709                 BLD #0,R1L ;Load bit 0 of R1L to C flag
35 SUB2_cod C 0016 182A                 SUBX.B R2H,R2L ;Subtraction
36 SUB2_cod C 0018 6709                 BST #0,R1L ;Store C flag to bit 0 of R1L
37 SUB2_cod C 001A 680A                 MOV.B R2L,R5 ;Store result
38 SUB2_cod C 001C 1805                 SUBS.W #1,R5 ;Decrement minuend address
39 SUB2_cod C 001E 1804                 SUBS.W #1,R4 ;Decrement subtrahend address
40 SUB2_cod C 0020 1A01                 DEC.B R1H ;Decrease byte counter
41 SUB2_cod C 0022 46EC                 BNE LOOP ;Branch if not R0L=0
42                                      ;
43 SUB2_cod C 0024 7709                 BLD #0,R1L ;Load bit 0 of R1L to C flag
44 SUB2_cod C 0026 06FB                 AMDC #H'FB,CCR ;Clear Z flag
45 SUB2_cod C 0028                      EXIT
46 SUB2_cod C 0028 5470                 RTS
47                                      ;
48                                      .END

*****TOTAL ERRORS   0
*****TOTAL WARNINGS 0
7.7 Addition of 8-Digit BCD Numbers

MCU: H8/300 Series
     H8/300L Series

Label name: ADDD1

7.7.1 Function

1. The software ADDD1 adds an 8-digit binary-coded decimal (BCD) number to another 8-digit 
   BCD number and places the result (an 8-digit BCD number) in a general-purpose register.
2. The arguments used with the software ADDD1 are unsigned integers.
3. All data is manipulated in general-purpose registers.

7.7.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Augend</td>
<td>R0, R1</td>
</tr>
<tr>
<td></td>
<td>Addend</td>
<td>R2, R3</td>
</tr>
<tr>
<td>Output</td>
<td>Result of addition</td>
<td>R0, R1</td>
</tr>
<tr>
<td></td>
<td>Carry</td>
<td>C flag (CCR)</td>
</tr>
</tbody>
</table>

7.7.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>†</td>
<td>†</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

I   U   H   U   N   Z   V   C

- UH UNZ VC

× : Unchanged
* : Indeterminate
† : Result
7.7.4 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>18</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>24</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

7.7.5 Description

1. Details of functions
   a. The following arguments are used with the software ADDD1:
      R0, R1: Contain an 8-digit BCD augend (32 bits long). After execution of the software ADDD1, the result of addition (an 8-digit BCD number, 32 bits long) is placed in this register.
      R2, R3: Contain an 8-digit BCD addend (32 bits long) as an input argument.
      C flag (CCR): Determines the presence or absence of a carry, as an output argument, after execution of the software ADDD1.
      C flag = 1: A carry occurred in the result of addition. (See figure 7.21.)
      C flag = 0: No carry occurred in the result of addition.
b. Figure 7.22 shows an example of the software ADDD1 being executed. When the input arguments are set as shown in (1), the result of addition is placed in R0 and R1 as shown in (2).
2. Notes on usage
   a. When upper bits are not used (see figure 7.23), set 0's in them; otherwise, no correct result can be obtained because addition is done on the numbers including indeterminate data.

   ![Figure 7.23 Example of Addition with Upper Bits Unused](image)

   b. After execution of the software ADDD1, the augend is destroyed because the result is placed in R1 and R2. If the augend is necessary after software ADDD1 execution, save it on memory.

3. Data memory
   The software ADDD1 does not use the data memory.
4. Example of use
Set an augend and an addend in the registers and call the software ADDD1 as a subroutine.

```
WORK1 .RES. W 2
MOV. W @WORK1, R0
MOV. W @WORK1+2, R1
JSR @ADDD1
OVER
```

```
WORK2 .RES. W 2
MOV. W @WORK2, R2
MOV. W @WORK2+2, R3
```

```
WORK3 .RES. W 2
MOV. W @WORK3, R0
MOV. W @WORK3+2, R1
```

```
REServes a data memory area in which the user program places an 8-digit BCD augend.
```

```
REServes a data memory area in which the user program places an 8-digit BCD addend.
```

```
REServes a data memory area in which the user program places the result of addition (an 8-digit BCD number).
```

```
Places in the input argument (R0, R1) the 8-digit BCD augend set by the user program.
```

```
Places in the input argument (R2, R3) the 8-digit BCD addend set by the user program.
```

```
Calls the software ADDD1 as a subroutine.
```

```
Branches to the carry processing routine if a carry has occurred in the result of addition.
```

```
Places the result (set in the output argument) in the data memory of the user program.
```

```
WORK1 .RES. W 2
WORK2 .RES. W 2
WORK3 .RES. W 2
MOV. W @WORK1, R0
MOV. W @WORK1+2, R1
MOV. W @WORK2, R2
MOV. W @WORK2+2, R3
BCS OVER
MOV. W R0, @WORK3
MOV. W R1, @WORK3+2
OVER Carry processing routine.
```

5. Operation
a. Addition of 2 bytes or more of BCD numbers can be done by performing a series of 1-byte additions with decimal correction.

b. A 1-byte add instruction (ADD.B), which does not involve the state of the C flag, is used to add the most significant byte given in equation 1. If a carry occurs after execution of equation 1, the C flag is set. Then a decimal correct instruction (DAA) is used to perform decimal correction.

\[
R1L + R3L \rightarrow R1L \quad \text{equation 1}
\]

Decimal correction of \(R1L \rightarrow R1L\)
c. A 1-byte add instruction (ADDX.B), which involves the state of the C flag, and a decimal-correct instruction are executed three times to add the upper bytes given in equation 2.

\[
\begin{align*}
R1H + R3H + C & \rightarrow R1H \quad \text{Decimal correction of } R1H \rightarrow R1H \\
R0H + R2L + C & \rightarrow R0L \quad \text{Decimal correction of } R0L \rightarrow R0L \\
R0H + R2H + C & \rightarrow R0H \quad \text{Decimal correction of } R0H \rightarrow R0H
\end{align*}
\]

\[\begin{array}{l}
\{ \text{equation 2} \}
\end{array}\]

The C flag indicates a carry that may occur in the result of addition of the least significant byte, the upper bytes of the lower word, and the lower bytes of the upper word that was executed in step (b).
7.7.6 Flowchart

```
ADDD1

R1L + R3L → R1L
   (Adds the least significant byte and places the result in R1L.)

Decimal correction of R1L

R1H + R3H + C → R1H
   (Performs decimal correction of the result.)

Decimal correction of R1H

R0L + R2L + C → R0L

Decimal correction of R0L

R0H + R2H + C → R0H

Decimal correction of R0H

RTS
```

Add the upper bytes involving a carry (the state of the C flag) that may occur in the result of addition of the lower bytes and sets the result in R1H, R0L and R0H with decimal correction.
7.7.7 Program List

*** H8/300 ASSEMBLER                VER 1.0B **   08/18/92 10:00:37
PROGRAM NAME =

1 ;********************************************************************
2 ;* 00 - NAME :DECIMAL ADDITION (ADDD1)
3 ;********************************************************************
4 ;*
5 ;********************************************************************
6 ;*
7 ;* ENTRY :R0 (UPPER WORD SUMMAND)
8 ;* R1 (LOWER WORD SUMMAND)
9 ;* R2 (UPPER WORD ADDEND)
10 ;* R3 (LOWER WORD ADDEND)
11 ;*
12 ;* RETURNS :R0 (UPPER WORD RESULT)
13 ;* R1 (LOWER WORD RESULT)
14 ;* C flag of CCR (C=0;TRUE , C=1;OVERFLOW)
15 ;*
16 ;********************************************************************

17 ADDD1_co C 0000 .SECTION ADDD1_code,CODE,ALIGN=2
18 .EXPORT ADDD1
19
20
21 ADDD1_co C 00000000 ADDD1 .EQU $ ;Entry point
22
23 ADDD1_co C 0000 0889 ADDD1_co C 0002 0F09 DAA RIL ;Decimal adjust RIL
24 ADDD1_co C 0004 0E31 ADDD1_co C 0006 0F01 DAA RIL ;Decimal adjust RIL
25 ADDD1_co C 0008 0E20 ADDD1_co C 000A 0F08 DAA ROL ;Decimal adjust ROL
26 ADDD1_co C 000C 0E20 ADDD1_co C 000E 0F08 DAA ROL ;Decimal adjust ROL
27 ADDD1_co C 0010 5470 RTS

28 ;
29 *****TOTAL ERRORS 0
30 *****TOTAL WARNINGS 0

31 ;
32 .END

136

RENESAS
7.8 Subtraction of 8-Digit BCD Numbers

MCU: H8/300 Series
    H8/300L Series

Label name: SUBD1

7.8.1 Function

1. The software SUBD1 subtracts an 8-digit binary-coded decimal (BCD) number from another 8-digit BCD number and places the result (an 8-digit BCD number) in a general-purpose register.
2. The arguments used with the software SUBD1 are unsigned integers.
3. All data is manipulated in general-purpose registers.

7.8.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Minuend</td>
<td>R0, R1</td>
<td>4</td>
</tr>
<tr>
<td>Subtrahend</td>
<td>R2, R3</td>
<td>4</td>
</tr>
<tr>
<td>Output Result of subtraction</td>
<td>R0, R1</td>
<td>4</td>
</tr>
<tr>
<td>Borrow</td>
<td>C flag (CCR)</td>
<td></td>
</tr>
</tbody>
</table>

7.8.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>†</td>
<td>†</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>•</td>
<td>×</td>
<td>•</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>†</td>
</tr>
</tbody>
</table>

×: Unchanged
•: Indeterminate
†: Result
### 7.8.4 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>18</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>24</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

### 7.8.5 Description

1. Details of functions
   a. The following arguments are used with the software SUBD1:
      - R0, R1: Contain an 8-digit BCD minuend (32 bits long). After execution of the software SUBD1, the result of subtraction (an 8-digit BCD number, 32 bits long) is placed in this register.
      - R2, R3: Contain an 8-digit BCD subtrahend (32 bits long) as an input argument.
      - C flag (CCR): Determines the presence or absence of a borrow, as an output argument, after execution of the software SUBD1.
      - C flag = 1: A borrow occurred in the result of subtraction. (See figure 7.24.)
      - C flag = 0: No borrow occurred in the result of subtraction.
b. Figure 7.25 shows an example of the software SUBD1 being executed. When the input arguments are set as shown in (1), the result of subtraction is placed in R0 and R1 as shown in (2).

![Diagram of subtraction with a borrow]

Figure 7.24  Example of Subtraction with a Borrow

Figure 7.25  Example of Software SUBD1 Execution
2. Notes on usage
   a. When upper bits are not used (see figure 7.26), set 0's in them; otherwise, no correct result
      can be obtained because subtraction is done on the numbers including indeterminate data.

   ![Figure 7.26 Example of Subtraction with Upper Bits Unused](image)

   b. After execution of the software SUBD1, the minuend is destroyed because the result is
      placed in R1 and R2. If the minuend is necessary after software SUBD1 execution, save it
      on memory.

3. Data memory
   The software SUBD1 does not use the data memory.
4. Example of use

Set a minuend and a subtrahend in the registers and call the software SUBD1 as a subroutine.

```
WORK1 .RES. W 2        Reserves a data memory area in which the user program places an 8-digit BCD minuend.
WORK2 .RES. W 2        Reserves a data memory area in which the user program places an 8-digit BCD subtrahend.
WORK3 .RES. W 2        Reserves a data memory area in which the user program places the result of subtraction (an 8-digit BCD number).

MOV. W @WORK1, R0      Places in the input argument (R0, R1) the 8-digit BCD minuend set by the user program.
MOV. W @WORK1+2, R1
MOV. W @WORK2, R2      Places in the input argument (R2, R3) the 8-digit BCD subtrahend set by the user program.
MOV. W @WORK2+2, R3

JSR @SUBD1             Calls the software SUBD1 as a subroutine.

BCS OVER

MOV. W R0, @WORK3      Branches to the borrow processing routine if a borrow has occurred in the result of subtraction.
MOV. W R1, @WORK3+2

OVER

Borrow processing routine.
```

5. Operation

a. Subtraction of 2 bytes or more of BCD numbers can be done by performing a series of 1-byte subtractions with decimal correction.

b. A 1-byte subtract instruction (SUB.B), which does not involve the state of the C flag, is used to add the most significant byte given in equation 1. If a borrow occurs after execution of equation 1, the C flag is set. Then a decimal correct instruction (DAS) is used to perform decimal correction.

\[ R1L - R3L \rightarrow R1L \quad \text{... equation 1} \]

Decimal correction of \( R1L \rightarrow R1L \)
c. A 1-byte subtract instruction (SUBX.B), which involves the state of the C flag, and a
decimal-correct instruction (DAS) are executed three times to add the upper bytes given in
equation 2.

\[
\begin{align*}
R1H - R3H - C & \rightarrow R1H \quad \text{Decimal correction of } R1H \rightarrow R1H \\
R0H - R2L - C & \rightarrow R0L \quad \text{Decimal correction of } R0L \rightarrow R0L \\
R0H - R2H - C & \rightarrow R0H \quad \text{Decimal correction of } R0H \rightarrow R0H
\end{align*}
\]
\[\text{equation 2}\]

The C flag indicates a borrow that may occur in the result of subtraction of the least
significant byte, the upper bytes of the lower word, and the lower bytes of the upper word
that was executed in step b. .
7.8.6  Flowchart

Subtracts the least significant byte and places the result in R1L.

Performs decimal correction of the result.

Subtracts the upper bytes involving a borrow (the state of the C flag) that may occur in the result of subtraction of the lower bytes and sets the result in R1H, R0L and R0H with decimal correction.
7.8.7 Program List

*** H8/300 ASSEMBLER    VER 1.0B  **   08/18/92 10:01:03

PROGRAM NAME =

;********************************************************************************
;* 00 - NAME :DECIMAL SUBTRACTION (SUBD1)
;********************************************************************************

; ENTRY :R0 (UPPER WORD MINUEND)
; R1 (LOWER WORD MINUEND)
; R2 (UPPER WORD SUBTRAHEND)
; R3 (LOWER WORD SUBTRAHEND)

; RETURNS :R0 (UPPER WORD RESULT)
; R1 (LOWER WORD RESULT)
; C flag OF CCR (C=0;TRUE,C=1;UNDER FLOW)

;********************************************************************************

; SUBD1_co C 0000  .SECTION SUBD1_code,CODE,ALIGN=2
; EXPORT SUBD1
; SUBD1_co C      00000000  SUBD1 .EQU $ ;Entry point
; SUBD1_co C 0000 18B9  SUB.B R3L,R1L ;R1L - R3L     -> R1L
; SUBD1_co C 0002 1F09  DAS R1L ;Decimal adjust R1L
; SUBD1_co C 0004 1E31  SUBX.B R3H,R1H ;R1H - R3H - C -> R1H
; SUBD1_co C 0006 1F01  DAS R1H ;Decimal adjust R1H
; SUBD1_co C 0008 1E20  SUBX.B R2L,R0L ;R0L - R2L - C -> R0L
; SUBD1_co C 000A 1F00  DAS R0L ;Decimal adjust R0L
; SUBD1_co C 000C 1E20  SUBX.B R2H,R0H ;R0H - R2H - C -> R0H
; SUBD1_co C 000E 1F00  DAS R0H ;Decimal adjust R0H
; RTS

; END

*****TOTAL ERRORS 0
*****TOTAL WARNINGS 0
7.9 Multiplication of 4-Digit BCD Numbers

MCU: H8/300 Series
    H8/300L Series

Label name: MULD

7.9.1 Function

1. The software MULD multiplies a 4-digit binary-coded decimal (BCD) number by another 4-digit BCD number and places the result (an 8-digit BCD number) in a general-purpose register.
2. The arguments used with the software MULD are unsigned integers.
3. All data is manipulated in general-purpose registers.

7.9.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplicand</td>
<td>R1</td>
<td>2</td>
</tr>
<tr>
<td>Multiplier</td>
<td>R0</td>
<td>2</td>
</tr>
<tr>
<td>Output</td>
<td>Result of multiplication</td>
<td>R2, R3</td>
</tr>
</tbody>
</table>

7.9.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5H</th>
<th>R5L</th>
<th>R6H</th>
<th>R6L</th>
<th>R7</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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

× : Unchanged
• : Indeterminate
† : Result
7.9.4 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>62</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>1192</td>
</tr>
</tbody>
</table>

Reentrant: Possible
Relocation: Possible
Interrupt: Possible

7.9.5 Notes

The clock cycle count (1192) in the specifications is for multiplication of 9999 by 9999.

7.9.6 Description

1. Details of functions
   a. The following arguments are used with the software MULD:
      R0: Contains a 4-digit BCD multiplier (16 bits long) as an input argument.
      R1: Contains a 4-digit BCD multiplicand (16 bits long) as an input argument.
      R2: Contains the upper 4 digits of the result (an 8-digit BCD, 32 bits long) as an output argument.
      R3: Contains the lower 4 digits of the result (an 8-digit BCD, 32 bits long) as an output argument.
b. Figure 7.27 shows an example of the software MULD being executed. When the input arguments are set as shown in (1), the result of multiplication is placed in R2 and R3 as shown in (2).

![Figure 7.27 Example of Software MULD Execution](image)

Table 7.3 lists the result of multiplication with 0's placed in input arguments.

**Table 7.3  Result of Multiplication with 0's Placed in Input Arguments**

<table>
<thead>
<tr>
<th>Input arguments</th>
<th>Output arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplicand</td>
<td>Multiplier</td>
</tr>
<tr>
<td>H'****</td>
<td>H'0000</td>
</tr>
<tr>
<td>H'0000</td>
<td>H'****</td>
</tr>
<tr>
<td>H'0000</td>
<td>H'0000</td>
</tr>
</tbody>
</table>

Note: H'**** is a hexadecimal number.
2. Notes on usage
   a. When upper bits are not used (see figure 7.28), set 0's in them; otherwise, no correct result can be obtained because multiplication is done on the numbers including indeterminate data placed in the upper bytes.

\[
\begin{array}{c}
R1 \\
0 & 0 & 8 & 6 \\
R0 \\
0 & 0 & 2 & 5 \\
R2 \\
0 & 0 & 0 & 0 \\
R3 \\
2 & 1 & 5 & 0
\end{array}
\]

Figure 7.28   Example of Multiplication with Upper Bits Unused

b. After execution of the software MULD, the multiplier is destroyed. If the multiplier is necessary after software MULD execution, save it on memory.

3. Data memory
   The software MULD does not use the data memory.
4. Example of use
Set a multiplicand and a multiplier in the registers and call the software MULD as a subroutine.

| WORK1  | .RES. W 1 | Reserves a data memory area in which the user program places a 4-digit BCD multiplicand. |
| WORK2  | .RES. W 1 | Reserves a data memory area in which the user program places a 4-digit BCD multiplier. |
| WORK3  | .RES. W 2 | Reserves a data memory area in which the user program places the result of multiplication (an 8-digit BCD number). |
|        |           | Places in the input argument (R1) the 4-digit BCD multiplicand set by the user program. |
|        | MOV. W @WORK1, R1 | |
|        | MOV. W @WORK2, R0 | Places in the input argument (R0) the 4-digit BCD multiplier set by the user program. |
|        | JSR @MULD | Calls the software MULD as a subroutine. |
|        | MOV. W R2, @WORK3 | Places the result (set in the output argument) in the data memory of the user program. |
|        | MOV. W R3, @WORK3+2 | |

---

![Renesas Logo]

149
5. Operation

a. Multiplication of decimal numbers can be done by performing a series of additions and shifts. Figure 7.29 shows an example of multiplication ($568 \times 1234$).

![Diagram of multiplication]

Figure 7.29 Example of Multiplication ($5678 \times 1234$)

Figure 7.29 indicates that a product is obtained by performing a series of shifting the result of multiplication and adding the multiplicand.

First, 4 bits (1 digit of the BCD) are taken out of the most significant byte of the multiplier and the multiplicand is added by that value. The result is shifted 4 bits (1 digit of the BCD). Next, 4 bits are taken out of the upper byte of the multiplier and the multiplicand is added by that value. The result is added to the previously obtained result. By performing this series of operations as many times as the number of digits of the BCD (that is, four times) the final result of multiplication can be obtained.

b. The program runs in the following steps:

(i) H'04 is placed in the H6H counter that indicates the number of digits of data.
(ii) The result of multiplication (R2 and R3) is cleared.
(iii) R2 and R3 are shifted 4 bits (1 digit of the BCD) to the left.
(iv) The multiplier is loaded in units of 4 bits (1 digit of the BCD) to R5, starting at its upper bytes. Branches to step (vi) if R5L is 0.
(v) Decimal addition of the multiplicand to R2 and R3 is performed by the value of R5L.
(vi) R6H is decremented.
(vii) Steps (iii) to (vi) are repeated until R6H reaches 0.
7.9.7 Flowchart

Places the upper 4 bits of the multiplier in R5L.
Shifts R2 and R3 (the result of multiplication) 4 bits to the left.

Places H'04 (the number of digits of data) in the R6H counter and clears R2 and R3 (the result of multiplication).

Branches if R5L (the upper 4 bits of the multiplier) is 0.

R6L - #1 → R6L

NO

R6L = 0

YES

NO

R5L = H'00

YES

Branches if R5L (the upper 4 bits of the multiplier) is 0.
Performs decimal addition of the multiplicand (R1) to R2 and R3 as many times as the count set in R5L.

Repeats this as many times as the number of digits of data.
7.9.8  Program List

*** HS/300 ASSEMBLER                VER 1.0B **   08/18/92 10:01:29

PROGRAM NAME =

1 ;********************************************************************
2 ;*
3 ;* 00 - NAME :DECIMAL MULTIPLICATION
4 ;*
5 ;*
6 ;********************************************************************
7 ;*
8 ;* ENTRY :R1 (MULTIPLICAND)
9 ;*
10 ;*
11 ;* RETURNS :R2 (UPPER WORD OF RESULT)
12 ;*
13 ;*
14 ;********************************************************************
15 ;*
16 MULD_code C 0000 .SECTION MULD_code,CODE,ALIGN=2
17 .EXPORT MULD
18 ;
19 MULD_code C 00000000
20 MULD_code C 0000 FE04
21 MULD_code C 0002 70200000
22 MULD_code C 0006 0023
23 MULD_code C 0008
24 MULD_code C 0008 FDE4
25 MULD_code C 000A FDE4
26 MULD_code C 000C
27 MULD_code C 000C 1008 .SECTION MULD_code,CODE,ALIGN=2
28 MULD_code C 000E 1200
29 MULD_code C 0010 1200
30 MULD_code C 0012 1008
31 MULD_code C 0014 1203
32 MULD_code C 0016 120A
33 MULD_code C 0018 1202
34 MULD_code C 001A 19E
35 MULD_code C 001C 46E
36 MULD_code C 001E ADD2
37 MULD_code C 0020 4714
38 MULD_code C 0022
39 MULD_code C 0022 098B
40 MULD_code C 0024 070B
41 MULD_code C 0026 0E13
42 MULD_code C 0028 0703
43 MULD_code C 002A 8A00
44 MULD_code C 002C 070A
45 MULD_code C 002E 8200
46 MULD_code C 0030 0902
47 MULD_code C 0032 851
48 MULD_code C 0034 46EC
49 MULD_code C 0036
50 MULD_code C 0036 1AE6
51 MULD_code C 0038 46EC
52 ;
53 MULD_code C 003A 5470
54 .END

*****TOTAL ERRORS  0
*****TOTAL WARNINGS  0

REnesas
7.10  Division of 8-Digit BCD Numbers

MCU: H8/300 Series
     H8/300L Series

Label name: DIVD

7.10.1  Function

1. The software DIVD divides an 8-digit binary-coded decimal (BCD) number by another 8-digit
   BCD number and places the result (an 8-digit BCD number) in a general-purpose register.
2. The arguments used with the software DIVD are unsigned integers.
3. All data is manipulated in general-purpose registers.

7.10.2  Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Dividend</td>
<td>R0, R1</td>
<td>4</td>
</tr>
<tr>
<td>Divisor</td>
<td>R2, R3</td>
<td>4</td>
</tr>
<tr>
<td>Output Result of division (quotient)</td>
<td>R0, R1</td>
<td>4</td>
</tr>
<tr>
<td>Result of division (remainder)</td>
<td>R4, R5</td>
<td>4</td>
</tr>
<tr>
<td>Error Z flag (CCR)</td>
<td>Z flag (CCR)</td>
<td>1</td>
</tr>
</tbody>
</table>

7.10.3  Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>†</td>
<td>†</td>
<td>•</td>
<td>•</td>
<td>†</td>
<td>†</td>
<td>×</td>
<td>•</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>•</td>
<td>×</td>
<td>•</td>
<td>×</td>
<td>†</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

×: Unchanged
•: Indeterminate
†: Result
### 7.10.4 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>84</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>1162</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

### 7.10.5 Notes

The clock cycle count (1162) in the specifications is for division of 99999999 by 9999.
7.10.6 Description

1. Details of functions
   a. The following arguments are used with the software DIVD:
      
      **R0**: Contains the upper 4 digits of an 8-digit BCD dividend (32 bits long). After execution of the software DIVD, the upper 4 digits of the result of division (quotient) are placed in this register.

      **R1**: Contains the lower 4 bits of the 8-digit BCD dividend (32 bits long). After execution of the software DIVD, the lower 4 digits of the result of division (quotient) are placed in this register.

      **R2**: Contains the upper 4 digits of an 8-digit BCD divisor as an input argument.

      **R3**: Contains the lower 4 digits of the 8-digit BCD divisor as an input argument.

      **R4**: The upper 4 digits of an 8-digit BCD remainder are placed in this register as an output argument.

      **R5**: The lower 4 digits of the 8-digit BCD remainder are placed in this register as an output argument.

      **Z flag (CCR)**: Determines the presence or absence of an error (division by 0) with the software DIVD as an output argument.

      - **Z flag = 1**: The divisor was 0, indicating an error.
      - **Z flag = 0**: The divisor was not 0.

   b. Figure 7.30 shows an example of the software DIVD being executed. When the input arguments are set as shown in (1), the result of division is placed in the registers as shown in (2).

   ![Figure 7.30 Example of Software DIVD Execution](image)
c. Table 7.4 lists the result of division with 0's placed in input arguments.

<table>
<thead>
<tr>
<th>Input arguments</th>
<th>Output arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dividend (R0, R1)</td>
<td>Divisor (R2, R3)</td>
</tr>
<tr>
<td>H'********</td>
<td>H'000000000</td>
</tr>
<tr>
<td>H'000000000</td>
<td>H'00000000</td>
</tr>
<tr>
<td>H'000000000</td>
<td>H'000000000</td>
</tr>
</tbody>
</table>

Note: H'**** is a hexadecimal number.

2. Notes on usage
   a. When upper bits are not used (see figure 7.31), set 0's in them; otherwise, no correct result can be obtained because division is done on the numbers including indeterminate data placed in the upper bits.

b. After execution of the software DIVD, the dividend is destroyed because the quotient is placed in R0 and R1. If the dividend is necessary after software DIVD execution, save it on memory.

3. Data memory
   The software DIVD does not use the data memory.
4. Example of use

Set a dividend and a divisor in the registers and call the software DIVD as a subroutine.

<table>
<thead>
<tr>
<th>WORK1</th>
<th>.RES. W 2</th>
<th>Reserves a data memory area in which the user program places an 8-digit BCD dividend.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORK2</td>
<td>.RES. W 2</td>
<td>Reserves a data memory area in which the user program places an 8-digit BCD divisor.</td>
</tr>
<tr>
<td>WORK3</td>
<td>.RES. W 2</td>
<td>Reserves a data memory area in which the user program places an 8-digit BCD quotient.</td>
</tr>
<tr>
<td>WORK4</td>
<td>.RES. W 2</td>
<td>Reserves a data memory area in which the user program places an 8-digit BCD remainder.</td>
</tr>
</tbody>
</table>

MOV. W @WORK1, R0  
MOV. W @WORK1+2, R1  
MOV. W @WORK2, R2  
MOV. W @WORK2+2, R3  

<table>
<thead>
<tr>
<th>JSR @DIVD</th>
<th>Calls the software DIVD as a subroutine.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ ERROR</td>
<td>Branches to the error (division by 0) processing routine if an error (division by 0) has occurred as a result of division.</td>
</tr>
</tbody>
</table>
| MOV. W R0, @WORK3  
MOV. W R1, @WORK3+2  
MOV. W R4, @WORK4  
MOV. W R5, @WORK4+2 | Places the result (set in the output argument) in the data memory of the user program. |
| ERROR Division-by-0 processing routine | |

```
MOV. W @WORK1, R0
MOV. W @WORK1+2, R1
MOV. W @WORK2, R2
MOV. W @WORK2+2, R3

JSR @DIVD

BEQ ERROR

MOV. W R0, @WORK3
MOV. W R1, @WORK3+2
MOV. W R4, @WORK4
MOV. W R5, @WORK4+2

ERROR Division-by-0 processing routine
```
5. Operation

a. Division of decimal numbers can be done by performing a series of subtractions.

Figure 7.32 shows an example of division (64733088 \div 5).

![Figure 7.32 Example of Division (64733088 \div 5)](image)

b. The program runs in the following steps:

(i) The dividend is shifted 4 bits (1 digit of the BCD) to the left to place the upper 4 bits of the dividend in the lower 4 bits of the result of division (remainder).

(ii) The divisor is subtracted from the dividend. Subtractions are repeated until the result becomes negative. The number of subtractions thus done is placed in the lower 4 bits (the least significant digit) of the dividend. (2)→(3)→(1) in figure 7.32) When the result has become negative, the divisor is added to the result (remainder) to return to the value before subtractions. (4) in figure 7.32)

(iii) The steps (i) to (ii) are repeated as many times as 8 digits.
7.10.7 Flowchart

DIVD

#H'0000 → R4
R4 → R5

R4 = R2
YES

R5 = R3
NO

NO

YES

EXIT

#H'08 → R6L

#H'04 → R6H

Shift R1L 1 bit left
Rotate R1H 1 bit left
Rotate R0L 1 bit left
Rotate R0H 1 bit left

Rotate R5L 1 bit left
Rotate R5H 1 bit left
Rotate R4L 1 bit left
Rotate R4H 1 bit left

R6H - #1 → R6H

YES

NO

R6H ≠ 0

Places the number of digits of the divisor data in R6L.

Cleans R4 and R5.

Exits if the divisor (R2, R3) is 0.

Places the upper 4 bits of the dividend in the lower 4 bits of the result (remainder).
Branches if the result of decimal subtraction is positive. If it is negative, performs decimal addition of the divisor (R2, R3) to the remainder (R4, R5).

---

R1L + #H'01 → R1L

- Adds H'01 to R1L.

C flag (CCR) = 0

---

R5L - R3L → R5L
Decimal correction of R5L
R5H - R3H - C → R5H
Decimal correction of R5H
R4L - R2L - C → R4L
Decimal correction of R4L
R4H - R2H - C → R4H
Decimal correction of R4H

---

R5L - R3L → R5L
Decimal correction of R5L
R5H - R3H - C → R5H
Decimal correction of R5H
R4L - R2L - C → R4L
Decimal correction of R4L
R4H - R2H - C → R4H
Decimal correction of R4H

---

R1L - #H'01 → R1L

---

Branches if the result of decimal subtraction is positive. If it is negative, performs decimal addition of the divisor (R2, R3) to the remainder (R4, R5) and subtracts H'01 from R1L.
Decrement R6L (number of digits).
Branches if R6L is not #H'00.
Places 0 in the Z flag.

R6L - #1 \rightarrow R6L

LBL2
YES

R6L = 0
NO

0 \rightarrow Z\text{ flag}

EXIT

RTS
7.10.8 Program List

*** 88/300 ASSEMBLER          VER 1.0B **  08/18/92 10:02:05  PAGE     1

PROGRAM NAME =

1 ;********************************************************************
2 ;*
3 ;* 00 - NAME :MULTIPLE-PRECISION DECIMAL DIVISION (DIVD)
4 ;*
5 ;********************************************************************

6 ;*
7 ; ENTRY :R2,R3 (DIVISOR)
8 ;*  R0,R1 (DIVIDEND)
9 ;*
10 ; RETURNS :R0,R1 (QUOTIENT)
11 ;*  R4,R5 (REZIDUAL)
12 ;*  Z flag OF CCR (Z=1;FALSE , Z=0;TRUE)
13 ;*
14 ;********************************************************************

15 ;
16 DIVD_cod C 0000  .SECTION  DIVD_code,CODE,ALIGN=2
17 .EXPORT  DIVD
18
19 DIVD_cod C 00000000  DIVD .EQUS  $ ;Entry point
20 DIVD_cod C 0000 70000000  MOV.W  H'0000,R4 ;Clear R4
21 DIVD_cod C 0004 0049  MOV.W  R4,R5 ;Clear R5
22 DIVD_cod C 0006 1043  CMP.W  R4,R2
23 DIVD_cod C 0008 4064  BNE  LBL1 ;Branch if Z=0
24 DIVD_cod C 000A 1053  CMP.W  R5,R3
25 DIVD_cod C 000C 4744  BEQ  EXIT ;Branch if Z=1 then exit
26 DIVD_cod C 000E  LBL1
27 DIVD_cod C 000E FE08  MOV.B #H'08,R6L ;Set bit counter1
28 DIVD_cod C 0010  LBL2
29 DIVD_cod C 0010 0A09  INC.B  R1L ;Increment R1L
30 DIVD_cod C 0012 18BD  SUB.B  R3L,R5L ;R5L - R3L     -> R5L
31 DIVD_cod C 0014 1F0D  DAS.B  R5L ;Decimal adjust R5L
32 DIVD_cod C 0016 1E35  CSEB.R  R2L,R4L ;R2L - R4L - C -> R2L
33 DIVD_cod C 0018 1E24  CSEB.R  R2H,R4H ;R2H - R4H - C -> R2H
34 DIVD_cod C 001A 1EAC  CSEB.R  R1L,R5L ;R1L - R5L - C -> R1L
35 DIVD_cod C 001C 1F04  CSEB.R  R1H,R5H ;R1H - R5H - C -> R1H
36 DIVD_cod C 001E 1A06  DEC.B  R6H ;Decrement bit counter2
37 DIVD_cod C 0020 44EC  BNE  LBL4 ;Branch if Z=0
38 DIVD_cod C 0022 0A09  INC.B  R1L ;Increment R1L
39 DIVD_cod C 0024 18BD  SUB.B  R3L,R5L ;R5L - R3L     -> R5L
40 DIVD_cod C 0026 1F0D  DAS.B  R5L ;Decimal adjust R5L
41 DIVD_cod C 0028 1E35  CSEB.R  R2L,R4L ;R2L - R4L - C -> R2L
42 DIVD_cod C 002A 1E24  CSEB.R  R2H,R4H ;R2H - R4H - C -> R2H
43 DIVD_cod C 002C 1EAC  CSEB.R  R1L,R5L ;R1L - R5L - C -> R1L
44 DIVD_cod C 002E 1F04  CSEB.R  R1H,R5H ;R1H - R5H - C -> R1H
45 DIVD_cod C 0030 1A09  DEC.B  R6L ;Decrement bit counter1
46 DIVD_cod C 0032 0A09  INC.B  R1L ;Increment R1L
47 DIVD_cod C 0034 18BD  SUB.B  R3L,R5L ;R5L - R3L     -> R5L
48 DIVD_cod C 0036 1F0D  DAS.B  R5L ;Decimal adjust R5L
49 DIVD_cod C 0038 1E35  CSEB.R  R2L,R4L ;R2L - R4L - C -> R2L
50 DIVD_cod C 003A 1E24  CSEB.R  R2H,R4H ;R2H - R4H - C -> R2H
51 DIVD_cod C 003C 1EAC  CSEB.R  R1L,R5L ;R1L - R5L - C -> R1L
52 ;
53 DIVD_cod C 003E 1F04  CSEB.R  R1H,R5H ;R1H - R5H - C -> R1H
54 DIVD_cod C 0040 1A09  DEC.B  R6L ;Decrement bit counter1
55 DIVD_cod C 0042 0A09  INC.B  R1L ;Increment R1L
56 DIVD_cod C 0044 18BD  SUB.B  R3L,R5L ;R5L - R3L     -> R5L
57 DIVD_cod C 0046 1F0D  DAS.B  R5L ;Decimal adjust R5L
58 DIVD_cod C 0048 1E35  CSEB.R  R2L,R4L ;R2L - R4L - C -> R2L
59 DIVD_cod C 004A 1E24  CSEB.R  R2H,R4H ;R2H - R4H - C -> R2H
60 DIVD_cod C 004C 1EAC  CSEB.R  R1L,R5L ;R1L - R5L - C -> R1L
61 DIVD_cod C 004E 1F04  CSEB.R  R1H,R5H ;R1H - R5H - C -> R1H
62 DIVD_cod C 0050 1A09  DEC.B  R6L ;Decrement bit counter1
63 DIVD_cod C 0052 0A09  INC.B  R1L ;Increment R1L

RENESEAS
64 DIVD_cod C 0050 06FB             ANDC.B #\'11111011,CCR ;Clear Z flag of CCR
65 DIVD_cod C 0052                     EXIT
66 DIVD_cod C 0052 5470                 RTS
67 ;
68 .END

*****TOTAL ERRORS 0
*****TOTAL WARNINGS 0
7.11 Addition of Multiple-Precision BCD Numbers

MCU: H8/300 Series
H8/300L Series

Label name: ADDD2

7.11.1 Function

1. The software ADDD2 adds a multiple-precision binary-coded decimal (BCD) number to another multiple-precision BCD number and places the result in the data memory where the augend was placed.

2. The arguments used with the software ADDD2 are unsigned integers, each being up to 255 bytes long.

7.11.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Augend and addend byte count</td>
<td>R0L</td>
<td>1</td>
</tr>
<tr>
<td>Start address of augend</td>
<td>R3</td>
<td>2</td>
</tr>
<tr>
<td>Start address of addend</td>
<td>R4</td>
<td>2</td>
</tr>
<tr>
<td>Output Start address of the result of addition</td>
<td>R3</td>
<td>2</td>
</tr>
<tr>
<td>Error Z flag (CCR)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Carry C flag (CCR)</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

7.11.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>†</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>†</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>†</td>
<td>×</td>
<td>†</td>
<td></td>
</tr>
</tbody>
</table>

× : Unchanged
• : Indeterminate
† : Result
7.11.4 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>44</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>7680</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

7.11.5 Notes

The clock cycle count (7680) in the specifications is for addition of 255 bytes to 255 bytes.

7.11.6 Description

1. Details of functions
   a. The following arguments are used with the software ADD2:
      R0L: Contains, as an input argument, the byte count of an augend and an addend in 2-digit hexadecimals.
      R3: Contains the start address of the augend in the data memory area. The start address of the result of addition is placed in this register after execution of the software ADD2.
      R4: Contains, as an input argument, the start address of the addend in the data memory area.
Z flag (CCR): Indicates an error in data length as an output argument.
Z flag = 0: The data byte count (R0L) was not 0.
Z flag = 1: The data byte count (R0L) was 0 (indicating an error).
C flag (CCR): Determines the presence or absence of a carry, as an output argument, after
execution of the software ADDD2.
C flag = 0: No carry occurred in the result of addition.
C flag = 1: A carry occurred in the result of addition (see figure 7.28).
b. Figure 7.33 shows an example of the software ADDD2 being executed. When the input
arguments are set as shown in 1., the result of addition is placed in the data memory area as
shown in 2..

<table>
<thead>
<tr>
<th>(1) Input arguments</th>
<th>(2) Output arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition data byte count R0L</td>
<td>Start address of result of addition R3 (H'F000)</td>
</tr>
<tr>
<td>R0L (H'04)</td>
<td>F', 0, 0, 0</td>
</tr>
<tr>
<td>Start address of augend R3</td>
<td>Z flag</td>
</tr>
<tr>
<td>R3 (H'F000)</td>
<td>0</td>
</tr>
<tr>
<td>Start address of addend R4</td>
<td>C flag</td>
</tr>
<tr>
<td>R4 (H'F100)</td>
<td>0</td>
</tr>
<tr>
<td>Data memory area</td>
<td>Data memory area</td>
</tr>
</tbody>
</table>

Augend:

| Start address of augend R3  | 0 | 4 |
| R3 (H'F000)                 | F' |

Addend:

| Start address of addend R4  | 1 | 0 | 0 | 0 |
| R4 (H'F100)                 | F' |

Augend + Addend:

| Start address of result of addition R3 (H'F000) |
| R3 (H'F000) |
| F', 0, 0, 0 |

Z flag:

| Z flag |
| 0 |

C flag:

| C flag |
| 0 |

Result of addition:

<table>
<thead>
<tr>
<th>Result of addition</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3 (H'F000)</td>
</tr>
<tr>
<td>9, 3, 3, 2, 6, 9, 2, 4</td>
</tr>
</tbody>
</table>

Figure 7.33 Example of Software ADDD2 Execution
Figure 7.34 shows an example of addition with a carry that occurred in the result.

![Addition with a Carry](image)

**Figure 7.34 Example of Addition with a Carry**

2. Notes on usage
   a. When upper bits are not used (see figure 7.35), set 0's in them. The software ADDD2 performs byte-based addition; if 0's are not set in the upper bits unused, no correct result can be obtained because the addition is done on the numbers including indeterminate data.

![Addition with Upper Bits Unused](image)

**Figure 7.35 Example of Addition with Upper Bits Unused**

b. After execution of the software ADDD2, the augend is destroyed because the result is placed in the data memory area where the augend was set. If the augend is necessary after software ADDD2 execution, save it on memory.

3. Data memory
   The software ADDD2 does not use the data memory.
4. Example of use
This is an example of adding 8 bytes of data. Set the start addresses of a byte count, an augend, and an addend in the registers and call the software ADDD2 as a subroutine.

<table>
<thead>
<tr>
<th>WORK1</th>
<th>.RES. B 1</th>
<th>Reserved a data memory area in which the user program places a byte count.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORK2</td>
<td>.RES. B 8</td>
<td>Reserved a data memory area in which the user program places an 8-byte (16-digit BCD) augend.</td>
</tr>
<tr>
<td>WORK3</td>
<td>.RES. B 8</td>
<td>Reserved a data memory area in which the user program places an 8-byte (16-digit BCD) addend.</td>
</tr>
</tbody>
</table>

MOV. B @WORK1, R0L  
Places in the input argument (R0L) the byte count set by the user program.

MOV. W #WORK2, R3  
Places in the input argument (R3) the start address of the augend set by the user program.

MOV. W #WORK3, R4  
Places in the input argument (R4) the start address of the addend set by the user program.

JSR @ADDD2  
Call the software ADDD2 as a subroutine.

BCS OVER  
Branches to the carry processing routine if a carry has occurred in the result of addition.

OVER  
Carry processing routine.

5. Operation
a. Addition of multiple-precision BCD numbers can be done by performing a series of 1-byte add instructions (ADDX.B) with decimal-correct instructions (DAA) as the augend and addend data are placed in registers, 2 digits in 1 byte.
b. The address of the least significant byte of the data memory area for the augend is placed in R3, and the address of the least significant byte of the data memory area for the addend in R4.
c. R1L that is used for saving the C flag is cleared.
d. The augend and addend are loaded in R2L and R2H respectively, byte by byte, starting at their least significant byte and then equation 1 is executed:

where the C flag indicates a carry that may occur in the result of addition of the lower bytes.

\[
\text{R2L (augend) + R2H (addend) + C} \rightarrow \text{R2L} \\
\text{Decimal correction of R2L} \rightarrow \text{R2L} \\
\text{R2L} \rightarrow @R3
\]  

\ldots \ldots \text{equation 1}

e. The result of (d) is placed in the data memory area for the augend.

f. R3, R4, and R0L are decremented each time the process d. to e. terminates. This processing is repeated until R0L reaches 0.
7.11.7 Flowchart

- Sets R5 to the address of the least significant byte of the augend and the start address of the addend (R4) to the address of the least significant byte.
- Clears R0H and copies the contents of R0L (the addend) to the byte counter (R1H).
- Clears R1L (where the C flag is to be saved) to 0.
- Exits if R0L reaches 0.
- Copies the start address of the augend (R3) to R5.
- Exits if R0L reaches 0.
- Sets R5 to the address of the least significant byte of the augend and the start address of the addend (R4) to the address of the least significant byte.

ADDD2

#H'00 → R0H
R0L → R1H

R0L = 0

YES

R3 → R5

R0 - #1 → R0

R0 + R5 → R5
R0 + R4 → R4

R0 - #1 → R0

R0L = 0

NO

EXIT

1

2
Adds the augend, addend and carry and places the result (decimally corrected) in the data memory area where the augend was placed.

- Sets bit 0 of R1L in the C flag.
- Clears the Z flag to 0.
- Repeats this process as many times as the byte count of the addition data.

Note: ADDD2 is the same as ADD2, SUB2, and SUBD2 except for the step surrounded by dotted lines.
7.11.8 Program List

*** 08/100 ASSEMBLER                VER 1.0B **   08/18/92 10:02:42

PROGRAM NAME = 1.0B

;********************************************************************
;* 00 - NAME :MULTIPLE-PRECISION DECIMAL ADDITION
;*  (ADDD2)
;*
;********************************************************************

;*
;ENTRY :R0L (BYTE COUNTER OF ADDITION DATA)
;*  R3 (START ADDRESS OF AUGEND)
;*  R4 (START ADDRESS OF ADDEND)
;*
;RETURNS :R3 (START ADDRESS OF RESULT)
;*  Z flag OF CCR (Z=0;TRUE,Z=1;FALSE)
;*  C flag OF CCR (C=0;TRUE,C=1;OVERFLOW)
;*
;********************************************************************

;*
;ADDD2_co C 0000
 ;.SECTION ADDD2_code,CODE,ALIGN=2
 ;.EXPORT ADDD2
 ;
 ; ADDD2_co C      00000000
 ; MOV.B #H'00,R0H ;Clear R0H
 ; MOV.B R0L,R1H ;Clear R1H
 ; BEQ EXIT ;Branch if Z=1 then exit
 ; MOV.W R3,R5
 ; MAIN
 ; SUBS.W #1,R0 ;Decrement R0
 ; ADD.W R0,R5 ;Set end address to summand pointer
 ; ADD.W R0,R4 ;Set end address to addend pointer
 ; MOV.B #H'00,R1L ;Clear R1L
 ; LOOP
 ; SUBS.W #1,R5 ;Decrement summand pointer
 ; ADD.W #1,R4 ;Decrement addend pointer
 ; DEC.B R1H ;Decrement R1H
 ; BNE LOOP ;Branch if Z=0
 ;
 ; ADDD2_co C 0010
 ; SUBS.W #1,R5 ;Decrement summand pointer
 ; BLD #0,R1L ;Load bit 0 of R1L to C flag
 ; ANDC.B #H'FB,CCR ;Clear Z flag of CCR
 ; EXIT
 ; RTS
 ;
 ;.END

***** TOTAL ERRORS 0
***** TOTAL WARNINGS 0
7.12 Subtraction of Multiple-Precision BCD Numbers

MCU: H8/300 Series
     H8/300L Series

Label name: SUBD2

7.12.1 Function

1. The software SUBD2 subtracts a multiple-precision binary-coded decimal (BCD) number from another multiple-precision BCD number and places the result in the data memory where the minuend was set.
2. The arguments used with the software SUBD2 are unsigned integers, each being up to 255 bytes long.

7.12.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Minuend and subtrahend byte count</td>
<td>R0L</td>
</tr>
<tr>
<td></td>
<td>Start address of minuend</td>
<td>R3</td>
</tr>
<tr>
<td></td>
<td>Start address of subtrahend</td>
<td>R4</td>
</tr>
<tr>
<td>Output</td>
<td>Start address of result</td>
<td>R3</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>Z flag (CCR)</td>
</tr>
<tr>
<td></td>
<td>Borrow</td>
<td>C flag (CCR)</td>
</tr>
</tbody>
</table>

7.12.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>†</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>†</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>†</td>
<td>×</td>
<td>†</td>
<td></td>
</tr>
</tbody>
</table>

×: Unchanged
•: Indeterminate
†: Result
7.12.4 Specifications

<table>
<thead>
<tr>
<th>Program memory (bytes)</th>
<th>44</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>7680</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

7.12.5 Notes

The clock cycle count (7680) in the specifications is for subtraction of 255 bytes from 255 bytes.

7.12.6 Description

1. Details of functions
   a. The following arguments are used with the software SUBD2:
      R0L: Contains, as an input argument, the byte count of a minuend and the byte count of a subtrahend in 2-digit hexadecimal.
      R3: Contains, as an input argument, the start address of the data memory area where the minuend is placed. After execution of the software SUBD2, the start address of the result is placed in this register.
      R4: Contains, as an input argument, the start address of the data memory area where the subtrahend is placed.
      Z flag (CCR): Indicates an error in data length as an output argument.
Z flag = 0: The data byte count (R0L) was not 0.
Z flag = 1: The data byte count (R0L) was 0, indicating an error.

C flag (CCR): Determines the presence or absence of a borrow after software SUBD2 execution as an output argument.
C flag = 0: No borrow occurred in the result.
C flag = 1: A borrow occurred in the result. (See figure 7.36)

b. Figure 7.36 shows an example of the software SUBD2 being executed. When the input arguments are set as shown in (1), the result of subtraction is placed in the data memory area as shown in (2).

![Figure 7.36 Example of Software SUBD2 Execution](image)

Figure 7.37 shows an example of subtraction with a borrow that has occurred in the result.
2. Notes on usage
   a. When upper bits are not used (see figure 7.38), set 0's in them. The software SUBD2 performs byte-based subtraction; if 0's are not set in the upper bits unused, no correct result can be obtained because the subtraction is done on the numbers including indeterminate data.

b. After execution of the software SUBD2, the minuend is destroyed because the result is placed in the data memory area where the minuend was set. If the minuend is necessary after software SUBD2 execution, save it on memory.

3. Data memory
   The software SUBD2 does not use the data memory.

4. Example of use
   This is an example of subtracting 8 bytes of data. Set the start addresses of a byte count, a minuend and a subtrahend in the registers and call the software SUBD2 as a subroutine.
5. Operation

a. Subtraction of multiple-precision binary numbers can be done by repeating a 1-byte subtract instruction (SUBX.B) and a decimal-correct instruction (DAA) as the minuend and subtrahend data are placed in registers, 2 digits in 1 byte.

b. The least significant byte of the data memory area for the minuend is placed in R3, and the least significant byte of the data memory area for the subtrahend in R4.

c. R1L that is used for saving the C flag is cleared.

d. The minuend and subtrahend are loaded in R2L and R2H respectively, byte by byte, starting at their least significant byte and equation 1 is executed:

\[
\begin{align*}
R2L (\text{minuend}) & - R2H (\text{subtrahend}) - C \rightarrow R2L \\
\text{Decimal correction of } R2L & \rightarrow R2L \\
R2L & \rightarrow @R3
\end{align*}
\]

where the C flag indicates a borrow that may occur in the result of subtraction of the lower bytes.

e. The result of d. is placed in the data memory area for the minuend.

f. R3, R4, and R0L are decremented each time the process d. to e. terminates. This processing is repeated until R0L reaches 0.
Sets R5 to the least significant byte address of the minuend and the start address of the subtrahend (R4) to the least significant byte address.

Clears R0H and copies the contents of R0H to the byte counter (R1H).

Exits if R0L reaches 0.

Copies the start address of the minuend (R3) to R5.

Sets R5 to the least significant byte address of the minuend and the start address of the subtrahend (R4) to the least significant byte address.

Clears R1L (where the C flag is to be saved) to 0.
Subtracts the subtrahend and borrow from the minuend, performs decimal correction of the result and places it in the data memory area where the minuend is placed.

Repeats this process as many times as the byte count of the subtraction data.

Sets bit 0 of R1L in the C flag.

Clears the Z flag to 0.

Note: SUBD2 is the same as ADD2, SUB2, and ADDD2 except for the step surrounded by dotted lines.
Program Name =

 intimidation: Name : Multiple-Precision Decimal Subtraction

 ENTRY : R0L (BYTE LENGTH OF DATA)
 R3 (START ADDRESS OF MINUEND)
 R4 (START ADDRESS OF SUBTRAHEND)
 RETUNS : R3 (START ADDRESS OF RESULT)
 Z BIT OF CCR (Z=0; TRUE , Z=1; FALSE)
 C BIT OF CCR (C=0; TRUE , C=1; OVERFLOW)

 ;********************************************************************
 ;********** SUBD2_code,CODE,ALIGN=2
 .EXPORT SUBD2

 ;ENTRY: R0L (BYTE LENGTH OF DATA)
 MOV.B #H00,R0H ;Clear R0H
 MOV.B R0L,R1H ;Set byte counter
 BEQ EXIT ;Branch if Z=1 then exit
 MOV.W R3,R5

 MAIN
 SUBD2_co C 0008 LOOP
 MOV.B #H00,R1L ;Clear R1L
 D SUBD2_co C 0010 SUBS.W #1,R0 ;Decrement byte length
 ADD.W R0,R5 ;Set end address of minuend
 ADD.W R0,R4 ;Set end address of substrahend
 MOV.B R0H,R1L ;Clear R1L

 LOOP
 MOV.B @R5,R2L ;Load minuend data
 MOV.B @R4,R2H ;Load substrahend data
 SUBX.B R2H,R2L ;R2L - R2H - C --> R2L
 DAS R2L ;Decimal adjust R2L
 BST #0,R1L ;Bit store bit 0 of R1L
 MOV.B R2L,R1L ;Store result

 SUBD2_co C 0010 DDS
 MOV.B R5,R2L ;Load minuend data
 MOV.B R4,R2H ;Load substrahend data
 SUBX.B R2H,R2L ;R2L - R2H - C --> R2L
 DAS R2L ;Decimal adjust R2L
 BST #0,R1L ;Bit store bit 0 of R1L
 MOV.B R2L,R1L ;Store result

 DDS
 MOV.B #H00,R1L ;Clear R1L
 RETS

 SEGMENT

 MOV.B #H00,R1L ;Clear R1L
 MOV.B R1L,R2L ;R2L = R1L
 DAS R2L ;Decimal adjust R2L
 BST #0,R1L ;Bit store bit 0 of R1L
 MOV.B R2L,R1L ;Store result

 DDS
 MOV.B #H00,R1L ;Clear R1L
 RETS

.EXIT

**** TOTAL ERRORS 0
**** TOTAL WARNINGS 0
7.13 Addition of Signed 32-Bit Binary Numbers

MCU: H8/300 Series
H8/300L Series

Label name: SADD

7.13.1 Function

1. The software SADD adds a signed 32-bit binary number to another signed 32-bit binary number and places the result in a general-purpose register.
2. The arguments used with the software SADD are signed integers.
3. All data is manipulated on general-purpose registers.

7.13.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Augend</td>
<td>R0, R1</td>
<td>4</td>
</tr>
<tr>
<td>Addend</td>
<td>R2, R3</td>
<td>4</td>
</tr>
<tr>
<td>Output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result of addition</td>
<td>R0, R1</td>
<td>4</td>
</tr>
<tr>
<td>Carry</td>
<td>V flag (CCR)</td>
<td>1</td>
</tr>
</tbody>
</table>

7.13.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>†</td>
<td>†</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>†</td>
<td>×</td>
</tr>
</tbody>
</table>

×: Unchanged
*: Indeterminate
†: Result
7.13.4 Specifications

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>20</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>44</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

7.13.5 Description

1. Details of functions
   a. The following arguments are used with the software SADD:
      (i) Input arguments:
          R0, R1: Contain a signed 32-bit binary augend.
          R2, R3: Contain a signed 32-bit binary addend.
      (ii) Output arguments
          R0, R1: Contain the result of addition (a signed 32-bit binary number)
          V flag (CCR): Determines the presence or absence of a carry as a result of addition.
          V flag = 1: A carry occurred in the result.
          V flag = 0: No carry occurred in the result.
b. Figure 7.39 shows an example of the software SADD being executed. When the input arguments are set as shown in 1., the result of addition is placed in R0 and R1 as shown in 2.

![Diagram of software SADD execution](image)

2. Notes on usage
   a. After execution of the software SADD, the augend is destroyed because the result is placed in R0 and R1. If the augend is necessary after software SADD execution, save it on memory.

3. Data memory
   The software SADD does not use the data memory.
4. Example of use

Set an augend and an addend in the input arguments and call the software SADD as a subroutine.

<table>
<thead>
<tr>
<th>WORK1</th>
<th>. RES. W 2</th>
<th>Reserves a data memory area in which the user program places a signed 32-bit binary augend.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORK2</td>
<td>. RES. W 2</td>
<td>Reserves a data memory area in which the user program places a 32-bit binary addend.</td>
</tr>
<tr>
<td>WORK3</td>
<td>. RES. W 2</td>
<td>Reserves a data memory area for storage of the result of addition.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Places in the input arguments (R0 and R1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the 32-bit binary augend set by the user program.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Places in the input arguments (R2 and R3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the 32-bit binary addend set by the user program.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Call the software SADD as a subroutine.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Places the result (set in the output arguments (R3 and R4)) in the data memory area of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the user program.</td>
</tr>
</tbody>
</table>

MOV. W @WORK1, R0
MOV. W @WORK1+2, R1
MOV. W @WORK2, R2
MOV. W @WORK2+2, R3

JSR @SADD

RES. W 2
VBS OVER
MOV. W R0, @WORK3
MOV. W R1, @WORK3+2

OVER Carry processing routine
5. Operation

a. Addition of signed 32-bit binary numbers is done by using add instructions (ADD.W and ADDX.B).

b. The addition steps are as follows:

(i) An augend is placed in R0 and R1 and an addend in R2 and R3.
(ii) The user bits (bits 6 and 4) and the overflow flag (bit 2) are cleared.
(iii) If the augend is negative, sign bit "1" is placed in the user bit (bit 6) of the CCR. If the addend is negative, sign bit "1" is placed in the user bit (bit 4) of the CCR.
(iv) The augend is added to the addend as follows:

\[
R1 + R3 \rightarrow R1 \\
R0L + R2L + C \rightarrow R0L \\
R0H + R2H + C \rightarrow R0H
\]

\[ \text{equation 1} \]

(v) Whether to continue processing or clear the V flag is determined depending on the state of the sign bit (CCR user bit):

<table>
<thead>
<tr>
<th>&lt;Sign bit&gt;</th>
<th>Bit 6 of CCR (Augend)</th>
<th>Bit 4 of CCR (Addend)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
7.13.6 Flowchart

SADD

- 0 → Bit 6 of CCR
- 0 → Bit 4 of CCR

Bit 7 of R0H → C flag

- C flag = 0
- NO
- 1 → Bit 6 of CCR

Bit 7 of R2H → C flag

- C flag = 0
- NO
- 1 → Bit 4 of CCR

1 → Bit 6 of CCR

R1 + R3 → R1

R0L + R2L + C → R0L

R0H + R2H + C → R0H

CCR → R6L

Bit 6 of R6L → C flag

Bit 4 of R6L

C flag → C flag

- C flag = 0

- Places the sign bit of the augend in bit 6 of CCR.

- Places the sign bit of the addend in bit 4 of CCR.

- Adds the augend and addend.

- Checks the code of the augend and addend.
1

YES

C flag = 0

NO

0 → V flag

......{ Clears the V flag.

LBL3

RTS
7.13.7 Program List

PROGRAM NAME =

;****************************************************************************
;* 00 - NAME :SIGNED 32 BIT BINARY ADDITION (SADD)
;****************************************************************************

ENTRY : R0 (UPPER WORD OF SUMMAND)
R1 (LOWER WORD OF SUMMAND)
R2 (UPPER WORD OF ADDEND)
R3 (LOWER WORD OF ADDEND)

RETURNS : R0 (UPPER WORD OF RESULT)
R1 (LOWER WORD OF RESULT)
V FLAG OF CCR

(V=0;TRUE,V=1:OVERFLOW OR UNDERFLOW)

;****************************************************************************

SADD_code C 0000

.SECTION SADD_code, CODE, ALIGN=2
.EXPORT SADD

SADD_code C      00000000
SADD .EQU $ ;Entry point
SADD_code C 0000 06AD
ANDC #H'AD,CCR ;Clear user bits and V flag of CCR
SADD_code C 0002 7770
BLD #7,R0H ;Load sign bit of summand
SADD_code C 0004 4402
BCC LBL1 ;Branch if C=0
SADD_code C 0006 0440
ORC.B #H'40,CCR ;Bit set user bit (bit 6 of CCR)
SADD_code C 0008
LBL1
SADD_code C 0008 7772
BLD #7,R2H ;Load sign bit of addend
SADD_code C 000A 4402
BCC LBL2 ;Branch if C=0
SADD_code C 000C 0410
ORC.B #H'10,CCR ;Bit set user bit (bit 4 of CCR)
SADD_code C 000E
LBL2
SADD_code C 000E 0931
ADDC R3,R1 ;R3 + R1
SADD_code C 0010 02E6
ADDC.R R2L,RSL ;R2L + RSL + C -> RSL
SADD_code C 0012 02E0
ADD.B R2H,RSH ;R2H + RSH + C -> RSH
SADD_code C 0014 020E
STC CCR,RSL ;CCR -> RSL
SADD_code C 0016 775E
BLD #6,RSL ;Bit load bit 4 of RSL
SADD_code C 0018 754E
SXOR #4,RSL ;Bit exclusive OR sign bits
SADD_code C 001A 4402
BCC LBL3 ;Branch if C=0
SADD_code C 001C 04FD
ANDC.B #H'FD,CCR ;Clear V flag
SADD_code C 001E
LBL3
SADD_code C 001E 5470
RTS

; 

****TOTAL ERRORS   0
****TOTAL WARNINGS  0
7.14 Multiplication of Signed 16-Bit Binary Numbers

MCU: H8/300 Series  
     H8/300L Series

Label name: SMUL

7.14.1 Function

1. The software SMUL multiplies a signed 16-bit binary number to another signed 162-bit binary number and places the result in a general-purpose register.
2. The arguments used with the software SMUL are signed integers.
3. All data is manipulated on general-purpose registers.

7.14.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>R1</td>
<td>2</td>
</tr>
<tr>
<td>Multiplicand</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplier</td>
<td>R0</td>
<td>2</td>
</tr>
<tr>
<td>Output</td>
<td>R1, R2</td>
<td>4</td>
</tr>
<tr>
<td>Result of multiplication</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7.14.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6H</th>
<th>R6L</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>†</td>
<td>†</td>
<td>×</td>
<td>×</td>
<td>•</td>
<td>•</td>
<td>×</td>
<td>•</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

×: Unchanged  
*: Indeterminate  
†: Result
7.14.4 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>52</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>132</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

7.14.5 Notes

The clock cycle count (132) in the specifications is a maximum cycle count.

7.14.6 Description

1. Details of functions
   a. The following arguments are used with the software SMUL:
      (i) Input arguments:
          R0: Contains a signed 16-bit binary multiplier.
          R1: Contains a signed 162-bit binary multiplicand.
      (ii) Output arguments
          R1, R2: Contain the result of multiplication (a signed 16-bit binary number)
   b. Figure 7.40 shows an example of the software SMUL being executed. When the input arguments are set as shown in (1), the result of multiplication is placed in R1 and R2 as shown in (2).
2. Notes on usage
   a. When upper bits are not used (see figure 7.41), set 0's in them; otherwise, no correct result can be obtained because multiplication is done on the numbers including indeterminate data placed in the upper bits. (The upper bits referred to here do not include sign bits.)
b. After execution of the software SMUL, the multiplicand is destroyed because the upper 2 bytes of the result are placed in R1. If the multiplicand is necessary after software SMUL execution, save it on memory.

3. Data memory
   The software SMUL does not use the data memory.

4. Example of use
   Set a multiplicand and a multiplier in the input arguments and call the software SMUL as a subroutine.

```
WORK1 . RES. W 2
        
WORK2 . RES. W 2
        
WORK3 . RES. W 2
        
MOV. W @WORK1, R1
        
MOV. W @WORK2, R0
        
JSR @SMUL
        
MOV. W R1, @WORK3
MOV. W R2, @WORK3+2

RES. W 2
```

5. Operation
   a. Subtraction of signed 16-bit binary numbers is done in one of the following manners depending on the signs of the multiplicand and multiplier:

<table>
<thead>
<tr>
<th>(Multiplicand)</th>
<th>(Multiplier)</th>
<th>(Process)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+)</td>
<td>(+)</td>
<td>Multiplied directly.</td>
</tr>
<tr>
<td>(+)</td>
<td>(−)</td>
<td>Multiplied with the sign of the multiplier inverted.</td>
</tr>
<tr>
<td>(−)</td>
<td>(+)</td>
<td>Multiplied with the sign of the multiplicand inverted.</td>
</tr>
<tr>
<td>(−)</td>
<td>(−)</td>
<td>Multiplied with the signs of both multiplicand and multiplier inverted.</td>
</tr>
</tbody>
</table>
b. The multiplication steps are as follows:

(i) A multiplicand is placed in R1 and a multiplier in R0.

(ii) The user bit (CCR) is cleared.

(iii) If the multiplicand is negative, its sign is inverted. If the multiplier is negative, its sign bit is inverted. Bits 6 and 4 of the CCR (user bits) are used as the sign bits of the multiplicand and multiplier, respectively. If the multiplicand or multiplier is negative, "1" is set in the corresponding user bit.

(iv) Multiplication is done with the software MUL.

(v) The CCR is transferred to R6L.

(vi) The result is modified or unmodified depending on the signs of the multiplicand and multiplier, as follows:

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Multiplier</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+)</td>
<td>(+)</td>
<td>The result is unmodified.</td>
</tr>
<tr>
<td>(+)</td>
<td>(–)</td>
<td>The result has its sign inverted.</td>
</tr>
<tr>
<td>(–)</td>
<td>(+)</td>
<td></td>
</tr>
<tr>
<td>(–)</td>
<td>(–)</td>
<td></td>
</tr>
</tbody>
</table>
SMUL

0 → bit 6 of CCR
0 → bit 4 of CCR

Bit 7 of R1H → C

YES

C = 0

NO

1 → Bit 6 of CCR

Logical reversal of R1H, R1L

R1L + #1 → R1L
R1H + #H'00 + C → R1H

Bit 7 of R0H → C

YES

C = 0

NO

1 → Bit 4 of CCR

Logical reversal of R0H, R0L

R0L + #1 → R0L
R0H + #H'00 + C → R0H

CLR

Branches if the multiplicand is positive.

Logic reversal of R1H, R1L

Branches if the multiplier is positive.

Logic reversal of R0H, R0L

Reverses the sign of the multiplier.

Clears the user bits (CCR).
MUL

CCR → R6L

Bit 6 of R6L → C

Bit 4 of R6L ⊕ C → C

C = 0

YES

Logical reversal of R1H, R1L, R2H, R2L

NO

R2L + #1 → R2L
R2H + #H'00 + C → R2H
R1L + #H'00 + C → R1L
R1H + #H'00 + C → R1H

LBL3

RTS

--- Performs multiplication.

--- Transfers the value of CCR to R6L.

--- Branches if the result is positive.

--- Reverses the sign of the result.
**Program List**

197

---

197
About Short Floating-Point Numbers

Formats of Short Floating-Point Numbers

1. Internal representation of short floating-point numbers

For purposes of this Application Note, the following formats of representation apply to short floating-point numbers (R = real number):

a. Internal representation for R = 0

\[
\begin{array}{cccccc}
31 & 30 & 29 & 28 & 27 & 26 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

All of the 32 bits are 0's.

b. Normalized format

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
S & \alpha & \beta & & & & & \\
\end{array}
\]

\(\alpha\) is an exponent whose field is 8 bits long. \(\beta\) is a mantissa whose field is 32 bits long. The value of R can be represented by the following equation (on conditions that 1\(\leq\alpha\leq254\):

\[
R = 2^\alpha \times 2^{2^{-127}} \times (1 + 2^{-1} \times \beta_{22} + 2^{-2} \times \beta_{21} + \ldots + 2^{-23} \times \beta_0)
\]

where \(\beta_i\) is the value of the i-th bit (0 \(\leq\) i \(\leq\) 22) and S is a sign bit.

c. Denormalized format

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
S & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

where \(\alpha\) is a mantissa whose field is 23 bits long. This format is used to represent a real number too small to be represented in the normal format. In this format, R can be represented by the following equation:

\[
R = 2^\alpha \times 2^{2^{-126}} \times (2^{-1} \times \beta_{22} + 2^{-2} \times \beta_{21} + \ldots + 2^{-23} \times \beta_0)
\]

d. Infinity

\[
\begin{array}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
S & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

where \(\beta\) is a mantissa whose field is 32 bits long. In this Application Note, however, the following rules apply if all exponents are 1's:

Positive infinity when \(S = 0\)

\(R = +\infty\)

Negative infinity when \(S = 1\)

\(R = -\infty\)
2. Example of internal representation

If $S = B'0$ (binary)

$a = B'10000011$ (binary)

$\beta = B'1011100\cdots0$ (binary)

Then the corresponding real number is as follow:

$$R = 2^0 \times 2^{131-127} \times (1 + 2^{-1} + 2^{-2} + 2^{-3} + \cdots + 2^{-5})$$

$$= 16 + 8 + 2 + 1 + 0.5 = 27.5$$

a. Maximum and minimum values

Up to the following absolute maximum value ($R_{\text{max}}$) and minimum value ($R_{\text{min}}$) can be represented:

$$R_{\text{MAX}} = 2^{254-127} \times (1 + 2^{-1} + 2^{-3} + 2^{-4} + \cdots + 2^{-5})$$

$$= 3.37 \times 10^{38}$$

$$R_{\text{MIN}} = 2^{-128} \times 2^{-23} = 2^{-140} \approx 1.40 \times 10^{-45}$$
7.15  Change of a Short Floating-Point Number to a Signed 32-Bit Binary Number

MCU:  H8/300 Series
          H8/300L Series

Label name: FKTR

7.15.1  Function

1. The software FKTR changes a short floating-point number (placed in a general-purpose register) to a signed 32-bit binary number.
2. "0" is output when the short floating-point number is "0".
3. When the short floating-point number is not less than $2^{31}$, a maximum value ($2^{31} - 1$ or $-2^{31}$) with the same sign as that number is output. When the short floating-point number is not more than $|1|$, "0" is output.

7.15.2  Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Short floating-point number</td>
<td>R0, R1</td>
</tr>
<tr>
<td>Output</td>
<td>Signed 32-bit binary number</td>
<td>R2, R3</td>
</tr>
</tbody>
</table>

7.15.3  Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>†</td>
<td>†</td>
<td>•</td>
<td>×</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>•</td>
<td>×</td>
<td>•</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

×: Unchanged
•: Indeterminate
†: Result
### 7.15.4 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>100</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>108</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

### 7.15.5 Notes

The clock cycle count (108) in the specifications is for the example shown in figure 7.42.

For the format of floating-point numbers, see "About Short Floating-point Numbers <Reference>.”

### 7.15.6 Description

1. Details of functions
   a. The following arguments are used with the software FKTR:
      (i) Input arguments:
          R0: Contains the upper 2 bytes of a short floating-point number.
          R1: Contains the lower 2 bytes of the short floating-point number.
      (ii) Output arguments
          R2: Contains the upper 2 bytes of a signed 32-bit binary number.
R3: Contains the lower 2 bytes of the signed 32-bit binary number.

b. Figure 7.42 shows an example of the software FKTR being executed. When the input arguments are set as shown in (1), the result of change is placed in R2 and R3 as shown in (2).

![Diagram](image)

**Figure 7.42  Example of Software FKTR Execution**

2. Notes on usage
   a. When the short floating-point number is "0" or not more than |1|, "0" is output.
   b. When the short floating-point number is not less than |2^31|, a maximum value with the same sign (H'7FFFFFFF or H'80000000) is output.
   c. After execution of the software FKTR, the input arguments placed in R0 and R1 are destroyed. If the input arguments are necessary after software FKTR execution, save them on memory.

3. Data memory
   The software FKTR does not use the data memory.
4. Example of use

Set a short floating-point number in the general-purpose register and call the software FKTR as a subroutine.

<table>
<thead>
<tr>
<th>WORK1</th>
<th>. DATA. W 2, 0</th>
<th>Reserves a data memory area in which the user program places a short floating-point number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORK2</td>
<td>. DATA. W 2, 0</td>
<td>Reserves a data memory area in which the user program places a signed 32-bit binary number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOV. B @WORK1, R0</td>
<td>Places in R0 and R1 the short floating-point number set by the user program.</td>
</tr>
<tr>
<td></td>
<td>MOV. W @WORK1+2, R1</td>
<td>Calls the software FKTR as a subroutine.</td>
</tr>
<tr>
<td></td>
<td>MOV. W R2, @WORK2</td>
<td>Places in R2 and R3 the signed 32-bit binary number set in the output argument.</td>
</tr>
<tr>
<td></td>
<td>MOV. W R3, @WORK2+2</td>
<td></td>
</tr>
</tbody>
</table>

5. Operation

a. The software FKTR takes the following steps to change the short floating-point number to a signed 32-bit binary number:

   (i) If the input argument is "0", then "0" is output.

   (ii) If the exponent part is smaller than "H'7F", then "0" is output.

   (iii) If the exponent part is not less than "H'9E", a maximum value with the same sign is output.

b. First, the input argument is checked.

   (i) If the input argument is "0", then "0" is output.

   (ii) If the exponent part is smaller than "H'7F", then "0" is output.

   (iii) If the exponent part is not less than "H'9E", a maximum value with the same sign is output.

c. Next, if the input argument is not "0" and its absolute value is not less than "1" (the exponent part = H'7F) and smaller than $2^{31}$ (the exponent part = H'9E), the following operations are performed:

   (i) The implicit MSB is set.

   (ii) The mantissa part (24 bits long) in which the implicit MSB is contained is shifted 1 bit to the left.

   (iii) R3 and R2 are rotated 1 bit to the left.

   (iv) Steps (ii) and (iii) are repeated as many times as "R0H+1".

   (v) A negative number is obtained by two's complement when the sign bit is negative.
FKTR

#H’0000 → R2
R2 → R3

R0 → R0

R0 = R0
NO

R1 → R1

R1 = 0
NO

Bit 7 of R0H →
Bit 7 of R5L

Bit 7 of R0L → C

Rotate R0H 1 bit left

R0H < H’7F
NO

---

R2 and R3 are cleared.

Branches if the input argument is "0".

Places the sign bit in R5L.

Places the exponent part in R0H.

Branches if the exponent part is smaller than "7F".
Branches if R0H is smaller than "H'1F". (Exponent part="H'9E")

Places the sign bit in the C bit and places maximum value with the same sign if R0H is not less than "H'1F".

Sets the implicit MSB.

Adds 1 to R0H.

Shifts 1 bit to the left the mantissa part (24 bits long) where the implicit MSB has been placed.

Rotates R3 and R2 1 bit to the left.

Decrement R0H until it reaches "0".
Takes on two's complement of the 32-bit binary number (placed in R2 and R3) to obtain a negative number.

Branches if the sign bit is "0" (a positive number).

Bit 0 of R5L → C

C → C

YES

C = 0

NO

Logical reversal of R2H, R2L, R3H, R3L

R3L + #1 → R3L
R3H + #H'00 + C → R3H
R2L + #H'00 + C → R2L
R2H + #H'00 + C → R2H

Takes on two's complement of the 32-bit binary number (placed in R2 and R3) to obtain a negative number.
7.15.8 Program List

*** H8/300 ASSEMBLER                VER 1.0B **   08/18/92 10:17:31
PROGRAM NAME =

;******************************************************************************
;* 00 - NAME :CHANGE FLOATING POINT TO 32 BIT BINARY
;*  (FKTR)
;******************************************************************************
; ENTRY :R0 (UPPER WORD OF FLOATING POINT)
;       R1 (LOWER WORD OF FLOATING POINT)
; RETURNS :R2 (UPPER WORD OF 32 BIT BINARY)
;          R3 (LOWER WORD OF 32 BIT BINARY)
;******************************************************************************

16 FKTR_code C 0000
   .SECTION FKTR_code,CODE,ALIGN=2
   .EXPORT FKTR

18
   19 FKTR_code C 00000000
   20 FKTR_code C 78020000
   21 FKTR_code C 0004 0023

22
   23 FKTR_code C 0006 0000
   24 FKTR_code C 0008 4004
   25 FKTR_code C 000A 0011
   26 FKTR_code C 000C 4754

27 FKTR_code C 000E
   .SECTION FKTR_code,CODE,ALIGN=2
   .EXPORT FKTR

28
   29 FKTR_code C 0010 670D
   30 FKTR_code C 0012 7778
   31 FKTR_code C 0014 1200
   32 FKTR_code C 0016 5737
   33 FKTR_code C 0018 1850
   34 FKTR_code C 001A 4546
   35 FKTR_code C 001C A01F
   36 FKTR_code C 001E 4518
   37 FKTR_code C 0020 770D
   38 FKTR_code C 0022 450A
   39 FKTR_code C 0024 79027FFF
   40 FKTR_code C 0026 78023FFF
   41 FKTR_code C 0028 4034
   42 FKTR_code C 002A
   43 FKTR_code C 002C 78020000
   44 FKTR_code C 0032 78030000
   45 FKTR_code C 0034 402A

46
   47 FKTR_code C 0038
   48 FKTR_code C 0038 7078
   49 FKTR_code C 003A 8001
   50 FKTR_code C 003C

51
   52 FKTR_code C 003E 1201
   53 FKTR_code C 0040 1208

54
   55 FKTR_code C 0042 120B
   56 FKTR_code C 0044 1203
   57 FKTR_code C 0046 120A
   58 FKTR_code C 0048 1202
   59 FKTR_code C 004A 1A00
   60 FKTR_code C 004C 402E

61
   62 FKTR_code C 004E 770D
   63 FKTR_code C 0050 4410
   64 FKTR_code C 0052 1702

207
05 FKTR_cod C 0054 170A NOT R2L
06 FKTR_cod C 0056 1703 NOT R3H
07 FKTR_cod C 0058 170B NOT R3L
08 FKTR_cod C 005A 8B01 ADD.B #H'01,R3L
09 FKTR_cod C 005C 9300 ADDX.B #H'00,R3H
0A FKTR_cod C 005E 9A00 ADDX.B #H'00,R2L
0B FKTR_cod C 0060 9200 ADDX.B #H'00,R2H
0C FKTR_cod C 0062 5470 LBL5
0D FKTR_cod C 0062 5470 RTS
0E ;
0F .END

*****TOTAL ERRORS 0
*****TOTAL WARNINGS 0
7.16 Change of a Signed 32-Bit Binary Number to a Short Floating-Point Number

MCU: H8/300 Series
     H8/300L Series

Label name: KFTR

7.16.1 Function

1. The software KFTR changes a signed 32-bit binary number (placed in a general-purpose register) to a short floating-point number.

7.16.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Signed 32-bit binary number</td>
<td>R0, R1</td>
<td>4</td>
</tr>
<tr>
<td>Output Short floating-point y number</td>
<td>R0, R1</td>
<td>4</td>
</tr>
</tbody>
</table>

7.16.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>†</td>
<td>†</td>
<td>·</td>
<td>·</td>
<td>×</td>
<td>×</td>
<td>·</td>
<td>·</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>·</td>
<td>·</td>
<td>×</td>
<td>·</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

×: Unchanged
·: Indeterminate
†: Result
### 7.16.4 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>98</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>346</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

### 7.16.5 Notes

The clock cycle count (346) in the specifications is for the example shown in figure 7.43.

For the format of floating-point numbers, see "About Short Floating-point Numbers <Reference>.”

### 7.16.6 Description

1. Details of functions
   a. The following arguments are used with the software KFTR:
      i. Input arguments:
         - R0: Contains the upper 2 bytes of a signed 32-bit binary number.
         - R1: Contains the lower 2 bytes of the signed 32-bit binary number.
(ii) Output arguments
   R2: Contains the upper 2 bytes of a short floating-point number.
   R3: Contains the lower 2 bytes of the short floating-point number.

b. Figure 7.43 shows an example of the software KFTR being executed. When the input
   arguments are set as shown in (1), the result of change is placed in R0 and R1 as shown in
   (2).

![Figure 7.43 Example of Software KFTR Execution](image)

2. Notes on usage
   a. After execution of the software KFTR, the signed 32-bit binary number is destroyed
      because the result of change is placed in R0 and R1. If the signed 32-bit binary number is
      necessary after software KFTR execution, save it on memory.

3. Data memory
   The software KFTR does not use the data memory.
4. Example of use

Set a signed 32-bit binary number in the general-purpose register and call the software KFTR as a subroutine.

<table>
<thead>
<tr>
<th>WORK1</th>
<th>. DATA. W 2, 0</th>
<th>Reserves a data memory area in which the user program places a signed 32-bit binary number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORK2</td>
<td>. DATA. W 2, 0</td>
<td>Reserves a data memory area in which the user program places a short floating-point number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV. W  @WORK1, R0</td>
<td>Places in R0 and R1 the signed 32-bit binary number set by the user program.</td>
<td></td>
</tr>
<tr>
<td>MOV. W  @WORK1+2, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSR  @KFTR</td>
<td>Calls the software KFTR as a subroutine.</td>
<td></td>
</tr>
<tr>
<td>MOV. W  R0, @WORK2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV. W  R1, @WORK2+2</td>
<td>Places in R0 and R1 the short floating-point number set in the output argument.</td>
<td></td>
</tr>
</tbody>
</table>

5. Operation

a. The software KFTR first checks whether the signed 32-bit binary number is positive or negative; if it is negative, the software takes two's complement of the number. Next, the software performs either of the following operations depending on whether the upper 8 bits are 'H'00' or not;

(i) When the upper 8 bits are not 'H'00", the exponent part is calculated and shifted to the right to obtain a 24-bit binary number.

(ii) When the upper 8 bits are 'H'00', the exponent part is calculated and shifted to the left to place "1" in the MSB of the lower 24 bits.

Finally, "H'7F" is added to the exponent part to obtain floating-point form.
7.16.7 Flowchart

- Branches if the signed 32-bit binary number is "0".
- Clears the work registers (R5 and R4H).
- Places the sign bit of the signed 32-bit binary number in the LSB of R5L.
- Takes two's complement if the signed 32-bit binary number is negative; branches if it is positive.
- Branches if R0H is "0".
214

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

- **R0H → R5H**: Copies the contents of R0H to R5H.
- **D'32 → R4L**: Places the counter's initial value D'32 in R4L.
- **R4L - #1 → R4L**: Repeats shifting R5H and decrementing R4L until "1" is placed in the C flag.
- **R4L → R4H**: Saves R4L in R4H.
- **Shift R0H 1 bit right**: Changes the 32-bit data to 24-bit data.

**Flowchart**

1. **LBL3**
   - Shift R5H 1 bit left
   - **R4L - #1 → R4L**
   - **C = 0**
     - **YES**
     - **NO**
   - **R4L → R4H**

2. **LBL4**
   - Shift R0H 1 bit right
   - Rotate R0L, R1H, R1L 1 bit right
   - **R4L - #1 → R4H**
   - **YES**
     - **Z = 0**
   - **NO**
Repeats this operation until "1" is placed in the C flag. Also decrements R4H as many times as the shift count.

Adding "H'7F" to R4H (getabaki representation ??????). Changes the exponent part to short floating-point format.

Places the sign bit in the MSB of R0H.
7.16.8  Program List

*** H8/300 ASSEMBLER        VER 1.0B  **  08/18/92  09:39:38

PROGRAM NAME =

1  ;***************************************************************************
2  ;* 00 - NAME : CHANGE 32 BIT BINARY TO FLOATING POINT
3  ;*    (KFTR)
4  ;***************************************************************************
5  
6  ;-entry : R0 (UPPER WORD OF 32 BIT BINARY)
7  ;*  R1 (LOWER WORD OF 32 BIT BINARY)
8  ;*  RETURNS : R0 (UPPER WORD OF FLOATING POINT)
9  ;*    R1 (LOWER WORD OF FLOATING POINT)
10  ;***************************************************************************
11  
12  KFTR_code C 0000                      .SECTION KFTR_code,CODE,ALIGN=2
13  .EXPORT KFTR
14  
15  KFTR_code C 00000000                  KFTR  .EQU $ ;Entry point
16  
17  KFTR_code C 0000 0D00                 MOV.W R0, R0
18  KFTR_code C 0002 4604                 BNE LBL1
19  KFTR_code C 0004 0D11                 MOV.W R1, R1
20  KFTR_code C 0006 4758                 BEQ LBL7 ;Branch if R0=R1=0
21  KFTR_code C 0008                      LBL1
22  KFTR_code C 0008 79050000             MOV.W #H'0000,R5 ;Clear R5
23  KFTR_code C 000C 0CD4                 MOV.B R5L,R4H ;Clear R4H
24  KFTR_code C 000E 7770                 BLD #7,R0H
25  KFTR_code C 0010 1700                 NOT R0H ;2's complement 32 bit binary
26  KFTR_code C 0012 1708                 NOT R0L
27  KFTR_code C 0014 1701                 NOT R1H
28  KFTR_code C 0016 1709                 NOT R1L
29  KFTR_code C 0018 1700                 NOT RSH ;2's complement 32 bit binary
30  KFTR_code C 001A 1708                 NOT ROL
31  KFTR_code C 0018 1701                 NOT RHL
32  KFTR_code C 001A 1709                 NOT R1L
33  KFTR_code C 001C 8901                 ADD.B #H'01,R1L
34  KFTR_code C 001E 0C00                 MOV.B R0H,R0H
35  KFTR_code C 0020 471A                 BEQ LBL5 ;Branch if R0H=0
36  KFTR_code C 0022 0C05                 MOV.B R0H,R5H
37  KFTR_code C 0024 4818                 MOV.B #D'32,R4H ;Set bit counter1
38  KFTR_code C 0026 1005                 SHLL.B R1L ;Change 32 bit binary to mantissa
39  KFTR_code C 0028 1201                 ROTXL.B R1H
40  KFTR_code C 002A 1308                 ROTER.B ROL
41  KFTR_code C 002C 140C                 DEC.B R4L ;Decrement R4L
42  KFTR_code C 002E 0C00                 MOV.B R4L,R4L ;Branch if C=0
43  KFTR_code C 0030 44FA                 MOV.B R4L,R4L ;Push R4L to R4H
44  KFTR_code C 0032 0C04                 MOV.B R4L,R4H ;Set bit counter2
45  KFTR_code C 0034 1400                 DEC.B R4H ;Decrement bit counter2
46  KFTR_code C 0036 248B                 MOV.W #D'24,B4H ;Set bit counter2
47  KFTR_code C 0038 0D09                 SELB.B R1L ;Change 32 bit binary to mantissa
48  KFTR_code C 003A 1201                 ROTER.B RIL
49  KFTR_code C 003C 1308                 ROTER.B RIL
50  KFTR_code C 003E 140C                 DEC.B R4L ;Decrement bit counter1
51  KFTR_code C 0040 4012                 BRA LBL6 ;Branch always
52  
53  KFTR_code C 0042                      LBL6
54  
55  KFTR_code C 0044 1404                 DEC.B R4H ;Decrement bit counter2
56  
57  
58  KFTR_code C 0046 F418                 MOV.W #D'24,B4H ;Set bit counter2
59  KFTR_code C 0048 1009                 SELB.B R1L ;Change 32 bit binary to mantissa
60  KFTR_code C 004A 1201                 ROTER.B RIL
61  KFTR_code C 004C 1308                 ROTER.B RIL
62  KFTR_code C 0050 140C                 DEC.B R4L ;Decrement bit counter1
63  KFTR_code C 0052 LBL5               ;
64 KFTR_cod C 004C 44F6  
65 KFTR_cod C 004E 1308  
66 KFTR_cod C 0050 1301  
67 KFTR_cod C 0052 1309  
68 KFTR_cod C 0054  
69 KFTR_cod C 0056 877F  
70 KFTR_cod C 0058 8778  
71 KFTR_cod C 005A 8778  
72 KFTR_cod C 005C 7700  
73 KFTR_cod C 005E 8770  
74 KFTR_cod C 0060  
75 KFTR_cod C 0062 8770  
76 KFTR_cod C 0064  
77 KFTR_cod C 0066  
78 KFTR_cod C 0068  
79 KFTR_cod C 006A 8770  
80 KFTR_cod C 006C 8770  
81 KFTR_cod C 006E  
82 KFTR_cod C 0070  
83 KFTR_cod C 0072  
84 KFTR_cod C 0074  
85 KFTR_cod C 0076  
86 KFTR_cod C 0078  
87 KFTR_cod C 007A  
88 KFTR_cod C 007C  
89 KFTR_cod C 007E  
90 KFTR_cod C 0080  
91 KFTR_cod C 0082  
92 KFTR_cod C 0084  
93 KFTR_cod C 0086  
94 KFTR_cod C 0088  
95 KFTR_cod C 008A  
96 KFTR_cod C 008C  
97 KFTR_cod C 008E  
98 KFTR_cod C 0090  
99 KFTR_cod C 0092  
100 KFTR_cod C 0094  
101 KFTR_cod C 0096  
102 KFTR_cod C 0098  
103 KFTR_cod C 009A  
104 KFTR_cod C 009C  
105 KFTR_cod C 009E  
106 KFTR_cod C 00A0  
107 KFTR_cod C 00A2  
108 KFTR_cod C 00A4  
109 KFTR_cod C 00A6  
110 KFTR_cod C 00A8  
111 KFTR_cod C 00AA  
112 KFTR_cod C 00AC  
113 KFTR_cod C 00AE  
114 KFTR_cod C 00B0  
115 KFTR_cod C 00B2  
116 KFTR_cod C 00B4  
117 KFTR_cod C 00B6  
118 KFTR_cod C 00B8  
119 KFTR_cod C 00BA  
120 KFTR_cod C 00BC  
121 KFTR_cod C 00BE  
122 KFTR_cod C 00C0  
123 KFTR_cod C 00C2  
124 KFTR_cod C 00C4  
125 KFTR_cod C 00C6  
126 KFTR_cod C 00C8  
127 KFTR_cod C 00CA  
128 KFTR_cod C 00CC  
129 KFTR_cod C 00CE  
130 KFTR_cod C 00D0  
131 KFTR_cod C 00D2  
132 KFTR_cod C 00D4  
133 KFTR_cod C 00D6  
134 KFTR_cod C 00D8  
135 KFTR_cod C 00DA  
136 KFTR_cod C 00DC  
137 KFTR_cod C 00DE  
138 KFTR_cod C 00E0  
139 KFTR_cod C 00E2  
140 KFTR_cod C 00E4  
141 KFTR_cod C 00E6  
142 KFTR_cod C 00E8  
143 KFTR_cod C 00EA  
144 KFTR_cod C 00EC  
145 KFTR_cod C 00EE  
146 KFTR_cod C 00F0  
147 KFTR_cod C 00F2  
148 KFTR_cod C 00F4  
149 KFTR_cod C 00F6  
150 KFTR_cod C 00F8  
151 KFTR_cod C 00FA  
152 KFTR_cod C 00FC  
153 KFTR_cod C 00FE  
154 KFTR_cod C 0100  
155 KFTR_cod C 0102  
156 KFTR_cod C 0104  
157 KFTR_cod C 0106  
158 KFTR_cod C 0108  
159 KFTR_cod C 010A  
160 KFTR_cod C 010C  
161 KFTR_cod C 010E  
162 KFTR_cod C 0110  
163 KFTR_cod C 0112  
164 KFTR_cod C 0114  
165 KFTR_cod C 0116  
166 KFTR_cod C 0118  
167 KFTR_cod C 011A  
168 KFTR_cod C 011C  
169 KFTR_cod C 011E  
170 KFTR_cod C 0120  
171 KFTR_cod C 0122  
172 KFTR_cod C 0124  
173 KFTR_cod C 0126  
174 KFTR_cod C 0128  
175 KFTR_cod C 012A  
176 KFTR_cod C 012C  
177 KFTR_cod C 012E  
178 KFTR_cod C 0130  
179 KFTR_cod C 0132  
180 KFTR_cod C 0134  
181 KFTR_cod C 0136  
182 KFTR_cod C 0138  
183 KFTR_cod C 013A  
184 KFTR_cod C 013C  
185 KFTR_cod C 013E  
186 KFTR_cod C 0140  
187 KFTR_cod C 0142  
188 KFTR_cod C 0144  
189 KFTR_cod C 0146  
190 KFTR_cod C 0148  
191 KFTR_cod C 014A  
192 KFTR_cod C 014C  
193 KFTR_cod C 014E  
194 KFTR_cod C 0150  
195 KFTR_cod C 0152  
196 KFTR_cod C 0154  
197 KFTR_cod C 0156  
198 KFTR_cod C 0158  
199 KFTR_cod C 015A  
200 KFTR_cod C 015C  
201 KFTR_cod C 015E  
202 KFTR_cod C 0160  
203 KFTR_cod C 0162  
204 KFTR_cod C 0164  
205 KFTR_cod C 0166  
206 KFTR_cod C 0168  
207 KFTR_cod C 016A  
208 KFTR_cod C 016C  
209 KFTR_cod C 016E  
210 KFTR_cod C 0170  
211 KFTR_cod C 0172  
212 KFTR_cod C 0174  
213 KFTR_cod C 0176  
214 KFTR_cod C 0178  
215 KFTR_cod C 017A  
216 KFTR_cod C 017C  
217
7.17  Addition of Short Floating-Point Numbers

MCU:  H8/300 Series
      H8/300L Series

Label name:  FADD

7.17.1  Function

1. The software FADD adds short floating-point numbers placed in four general-purpose registers
   and places the result of addition in two of the four general-purpose registers.
2. All arguments used with the software FADD are represented in short floating-point form.

7.17.2  Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Augend</td>
<td>R0, R1</td>
<td>4</td>
</tr>
<tr>
<td>Addend</td>
<td>R2, R3</td>
<td>4</td>
</tr>
<tr>
<td>Output</td>
<td>Result of addition</td>
<td>R0, R1</td>
</tr>
</tbody>
</table>

7.17.3  Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>✧</td>
<td>✧</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✧</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>✧</td>
<td>✧</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

×  : Unchanged
•  : Indeterminate
✧  : Result
### 7.17.4 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>280</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>268</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

### 7.17.5 Notes

The clock cycle count (268) in the specifications is for the example shown in figure 7.44.

For the format of floating-point numbers, see "About Short Floating-point Numbers <Reference>."
7.17.6 Description

1. Details of functions
   a. The following arguments are used with the software FADD:
      (i) Input arguments:
         R0: Contains the upper 2 bytes of a short floating-point augend.
         R1: Contains the lower 2 bytes of the short floating-point augend.
         R2: Contains the upper 2 bytes of a short floating-point addend.
         R3: Contains the lower 2 bytes of the short floating-point addend.
      (ii) Output arguments
         R0: Contains the upper 2 bytes of the result.
         R1: Contains the lower 2 bytes of the result.
   b. Figure 7.44 shows an example of the software FADD being executed. When the input
      arguments are set as shown in (1), the result of addition is placed in R0 and R1 as shown in
      (2).

   ![Figure 7.44 Example of Software FADD Execution](image)

2. Notes on usage
   a. The maximum and minimum values that can be handled by the software FADD are as
      follows:
      - Positive maximum: H'7F800000
      - Positive minimum: H'00000001
      - Negative maximum: H'80000001
      - Negative minimum: H'FF800000
   b. All positive short floating-point numbers H'7F800000 to H'7FFFFFFF are treated as a
      maximum value (H'7F800000). All negative short floating-point numbers H'FF800000 to
      H'FFFFFFFF are treated as a minimum value (H'FF800000).
c. As a maximum value is treated as infinity (∞), the result of ∞ + 100 or ∞ - 100 becomes infinite. (See table 7.5.)

Table 7.5  Examples of Operation with Maximum Values Used as Arguments

<table>
<thead>
<tr>
<th>Augend</th>
<th>Addend</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>7F800000 to 7FFFFFFF</td>
<td>*******</td>
<td>7F800000</td>
</tr>
<tr>
<td>7F800000 to FFFFFFFF</td>
<td>7F800000 to 7FFFFFFF</td>
<td>7F800000</td>
</tr>
<tr>
<td>FF800000 to FFFFFFFF</td>
<td>*******</td>
<td>FF800000</td>
</tr>
<tr>
<td>7F800000 to 7FFFFFFF</td>
<td>FF800000 to FFFFFFFF</td>
<td>FF800000</td>
</tr>
</tbody>
</table>

Note: * represents a hexadecimal number.

d. H'80000000 is treated as H'00000000 (zero).
e. After execution of the software FADD, the augend and addend data are destroyed. If the input arguments are necessary after software FADD execution, save them on memory.

3. Data memory

The software FADD does not use the data memory.

4. Example of use

Set an augend and an addend in the general-purpose register and call the software FADD as a subroutine.

```
WORK1 . RES. W 2
WORK2 . RES. W 2
WORK3 . RES. W 2

RESERVES A DATA MEMORY AREA IN WHICH THE USER PROGRAM PLACES AN AUGEND.
RESERVES A DATA MEMORY AREA IN WHICH THE USER PROGRAM PLACES AN ADDEND.
RESERVES A DATA MEMORY AREA IN WHICH THE USER PROGRAM PLACES THE RESULT OF ADDITION.

RESERVES A DATA MEMORY AREA IN WHICH THE USER PROGRAM PLACES R0 AND R1 THE AUGEND SET BY THE USER PROGRAM.
RESERVES A DATA MEMORY AREA IN WHICH THE USER PROGRAM PLACES R2 AND R3 THE ADDEND SET BY THE USER PROGRAM.

CALLS THE SOFTWARE FADD AS A SUBROUTINE.

MOV. W R0, @WORK3
MOV. W R1, @WORK3+2

RESERVES A DATA MEMORY AREA IN WHICH THE USER PROGRAM PLACES R0 AND R1 THE RESULT OF ADDITION SET IN THE OUTPUT ARGUMENT.
```
5. Operation

Addition of short floating-point numbers is done in the following steps:

a. The software checks whether the augend and addend are +Åá or -Åá.
   (i) When the exponent part of the augend is H'FFF, either of the following values is output depending on the state of the sign bit:

<table>
<thead>
<tr>
<th>Sign bit</th>
<th>Output value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (positive)</td>
<td>H'7F800000 (+)</td>
</tr>
<tr>
<td>1 (negative)</td>
<td>H'FF800000 (–)</td>
</tr>
</tbody>
</table>

   (ii) The table shown in (a)-(i) also applies when the augend is neither +∞ nor –∞ and the exponent part of the addend is H'FF.

b. The software checks whether the augend and addend are "0".
   (i) If either the augend or addend is "0", the other number is output. (If both are "0", "H'00000000" is output.)

c. The software attempts to match the exponent part of the augend with that of the addend.
   (i) The smaller number of the exponent part is incremented and, at the same time, the mantissa part (including the implicit MSB) is shifted digit by digit to the right until the exponent part of the augend matches that of the addend. (In the case of the denormalized format, 1 is added to the exponent part so that the implicit MSB of the mantissa part is treated as "0".

d. The mantissa part of the augend is added to that of the addend.

e. The result of addition is represented in floating-point format.
(Example)
Augend
Sign bit=0, exponent part=H'F1, mantissa part=H'1ABCDE
(excluding the implicit MSB)
Addend
Sign bit=0, exponent part=H'F4, mantissa part=H'1B3DD2
(excluding the implicit MSB)

```
<table>
<thead>
<tr>
<th>Augend</th>
<th>Addend</th>
<th>Result of addition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 1 0 0 1</td>
<td>1 0 0 1 1 0 1 0 1 1 1 0 0 1 1 0 1 1 1 0</td>
<td>1 1 1 1 0 1 0 0 1 0 0 1 1 1 0 1 0 1 1 1 0 1 1 0 1 1 0 1</td>
</tr>
<tr>
<td>H'F1</td>
<td>H'9ABCDE</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1 0 1 0 0</td>
<td>1 0 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 0 1 0 1 0 0 1 0</td>
<td>1 1 1 1 0 1 0 0 1 0 0 1 1 1 0 1 0 1 1 1 0 1 1 0 1 0 1 0</td>
</tr>
<tr>
<td>H'F4</td>
<td>H'9B3CC2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The exponent part remains unchanged. Only the mantissa part undergoes addition.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result of addition = 1.36393511295×2⁻¹⁷</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(H'7A2E956D)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sign bit=0, exponent part=H'F4, mantissa part=H'2E956D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(excluding the implicit MSB)</td>
</tr>
</tbody>
</table>
```
7.17.7 Flowchart

FADD

#H'00 → R6L
#H'7F80 → R5

Bit 7 of R0H → Bit 0 of R6L

0 → Bit 7 of R0H

Bit 7 of R2H → Bit 1 of R6L

0 → Bit 7 of R2H

R0 ≥ R5

YES

NO

R2 < R5

YES

NO

Shift R6L 1 bit right

LBL4

LBL1

Bit 0 of R6L → C

C = 1

YES

NO

LBL2

LBL3

#H'7F80 → R0
#H'0000 → R1

#H'FF80 → R0
#H'0000 → R1

RTS

RTS

Clears R6L to 0. Places #H'7F80 in R5.

Places the sign bit of the augend in bit 0 of R6L.

Clears the sign bit of the augend.

Places the sign bit of the addend in bit 1 of R6L.

Clears the sign bit of the addend.

Branches if the exponent part of the augend is "H'FF".

Branches if the exponent part of the augend is not "H'FF".

Shifts the sign bit of the addend to bit 0 of R6L.

Places "H'7F800000" as output if the sign bit is "0" (positive), or "H'FF800000" as output if the sign bit is "1" (negative).
- LBL4
  - R1 → R1
  - YES
  - R1 ≠ 0
    - NO
    - R0 → R0
    - YES
    - R0 ≠ 0
      - NO
      - 1 → Bit 7 of R6L
        - 0 → Bit 0 of R6L

- LBL5
  - R3 → R3
  - YES
  - R3 ≠ 0
    - NO
    - R2 → R2
    - YES
    - R2 ≠ 0
      - NO
      - 1 → Bit 6 of R6L
        - 0 → Bit 1 of R6L

- LBL6
  - Bit 7 of R6L → C
  - C ∨ Bit 6 of R6L → C

- LBL7
  - YES
  - C = 0
    - NO
    - R1 + R3 → R1
      - R0 + R2 → R0
      - Bit 0 of R6L → C
        - C ∨ Bit 1 of R6L → C
          - C → Bit 7 of R0H

- RTS

---

Places "1" in bit 7 of R6L if the augend is "0".

Places "1" in bit 6 of R6L if the addend is "0".

Bit 7 of R6L is ANDed with bit 6 of R6L.
Branches if C = "0" (augend ≠ "0" addend ≠ "0").

Places in R0 and R1 the augend or addend as an output value..
Places "1" in bit 7 of R2L if the addend is represented in normalized format (R2H ≠ 0), and adds #1 to R2H if the addend is represented in denormalized format (R2H=0).

Places "1" in bit 7 of R0L if the augend is represented in normalized format (R0H ≠ 0), and adds #1 to R0H if the augend is represented in denormalized format (R0H=0).

Clears bit 7 of R0L.

Clears bit 7 of R2L.

Places the exponent part of the augend in R0H.

Places the exponent part of the addend.

Places "1" in bit 7 of R2L if the addend is represented in normalized format (R2H=0), and adds #1 to R2H if the addend is represented in denormalized format (R2H=0).
227

Shifts the mantissa of the addend to the right as many times as $R5H$ (the difference between exponents).

Finds the difference ($R5H$) if $R5H > R5L$.

Clears the augend to 0 and branches to (4) if $R5H > \#D'24$.

Places the exponent ($R0H$) of the augend in $R5H$ and the exponent ($R2H$) of the addend in $R5L$.

Compares $R5H$ with $R5L$: branches (4) to if $R5H = R5L$ or to (5) if $R5H < R5L$.

Shifts the mantissa of the addend to the right as many times as $R5H$ (the difference between exponents).

$R0H \rightarrow R5H$
$R2H \rightarrow R5L$

$R5H = R5L$

$R5H < R5L$

$R5H - R5L \rightarrow R5H$

$R5H < \#D'24$

$\#H'0000 \rightarrow R2$
$R2 \rightarrow R3$

$YES$

$NO$

$R5H \neq 0$

$YES$

$NO$

LBL16

LBL16

LBL16

lbl12

LBL14

LBL16

LBL16

LBL13

LBL16
Shifts the mantissa of the augend to the right as many as R5L (the difference between exponents).

Transfers R2H to R0H, clears the mantissa of the augend, and branches if R5L > #D'24.

Shifts the mantissa of the augend to the right as many as R5L (the difference between exponents).

Transfers R2H to R0H.

Checks the sign bits, and branches if they have opposite signs.
R1 + R3 → R1
R0L + R2L + C → R0L

Adds the mantissas.

C = 0

YES

NO

Rotate R0L, R1H and R1L 1 bit right

R0H + #1 → R0H

Rotates the mantissa 1 bit to the right and adds #1 to the exponent if a carry occurs.

R0H ≠ #H'FF

YES

NO

Branches to (10) if R0H ≠ #H'FF, or to (11) if R0H = #H'FF.

LBL1
Reverses the sign bit and takes two's complement of the mantissa if a borrow occurs.

Subtracts the mantissa.

Places H'00 in R0H and exits the program if the result of subtraction is "0".

Reverses the sign bit and takes two's complement of the mantissa if a borrow occurs.
R0H = 0

YES

NO

R0H - #1 → R0H

YES

C = 0

NO

C = 1

YES

Shift R0L 1 bit left
Rotate R1H and R0L 1 bit left

NO

R0H + #1 → R0H

LBL19

LBL20

LBL21

LBL22

LBL23

Shift R0H 1 bit right
C → Bit 7 of R0L

C → Bit 7 of R0H

Bit 0 of R6L → C

Shifts the mantissa to the left and decrements the exponent until a carry occurs.

Increments the exponent.

Shifts the mantissa 1 bit to the right.

Changes to normalized format if C=1, or denormalized format if C=0.

Changes to short floating-point format.
7.17.8 Program List

*** H8/300 ASSEMBLER ***
** VER 1.0B ** 08/18/92 10:20:43

PROGRAM NAME =

1 ;******************************************************************************
2 ;* 00 - NAME : FLOATING POINT ADDITION (FADD)
3 ;******************************************************************************
4 ;
5 ;******************************************************************************
6 ;* ENTRY : R0 (UPPER WORD OF SUMMAND)
7 ;* R1 (LOWER WORD OF SUMMAND)
8 ;* R2 (UPPER WORD OF ADDEND)
9 ;* R3 (LOWER WORD OF ADDEND)
10 ;
11 ;* RETURNS : R0 (UPPER WORD OF RESULT)
12 ;* R1 (LOWER WORD OF RESULT)
13 ;
14 ;******************************************************************************
15 ;
16 ;
17 FADD_code C 0000 .SECTION FADD_code,CODE,ALIGN=2
18 .EXPORT FADD
19 ;
20 FADD_code C 00000000 FADD .EQU $ ;Entry point
21 FADD_code C 0000 FADD MOV.B #H'00,R6L ;Clear REL
22 FADD_code C 0002 79057F80 MOV.W #H'7F80,R5 ;Set "H'7F80"
23 ;
24 FADD_code C 0006 7770 BLD #7,R0H
25 FADD_code C 0008 670E BST #0,R6L ;Set sign bit to bit 0 of R6L
26 FADD_code C 000A 7270 BCLR #7,R0H ;Bit clear bit 7 of R0H
27 ;
28 FADD_code C 000C 7772 BLD #7,R2H
29 FADD_code C 000E 671E BST #1,R6L ;Set sign bit to bit 1 of R6L
30 FADD_code C 0010 7272 BCLR #7,R2H ;Bit clear bit 7 of R2H
31 ;
32 FADD_code C 0012 1D05 CMP.W R0,R5
33 FADD_code C 0014 4306 BLS LBL1 ;Branch if "exponent of summand"="H'FF"
34 FADD_code C 0016 1D25 CMP.W R2,R5
35 FADD_code C 0018 421A BHI LBL4 ;Branch if not "exponent of summand"="H'FF"
36 FADD_code C 001A 110E SHLR R6L ;Shift R6L 1 bit right
37 FADD_code C 001C LBL1
38 FADD_code C 001C 770E BLD #0,R6L ;Bit load sign bit
39 FADD_code C 001E 450A BCS LBL3 ;Branch if sign bit=1
40 FADD_code C 0020 LBL2
41 FADD_code C 0020 79057F80 MOV.W #H'7F80,R0 ;Set plus maximum number
42 FADD_code C 0024 79010000 MOV.W #H'0001,R1
43 FADD_code C 0028 5470 RTS
44 FADD_code C 002A LBL3
45 FADD_code C 002A 79057F80 MOV.W #H'7F80,R0 ;Set minus minimum number
46 FADD_code C 002E 79010000 MOV.W #H'0001,R1
47 FADD_code C 0032 5470 RTS
48 ;
49 FADD_code C 0034 LBL4
50 FADD_code C 0034 0D11 MOV.W R1,R1 ;
51 FADD_code C 0036 4608 BNE LBL5 ;Branch if Z=0
52 FADD_code C 0038 0D00 MOV.W R0,R0 ;Branch if Z=0
53 FADD_code C 003A 4604 BNE LBL5 ;Branch if Z=0
54 FADD_code C 003C 707E BSET #7,R6L ;Bit set bit 7 of REL
55 FADD_code C 003E 720E BCLR #0,R6L ;Bit clear bit 0 of REL
56 FADD_code C 0040 LBL5
57 FADD_code C 0040 0D33 MOV.W R3,R3 ;
58 FADD_code C 0042 4608 BNE LBL6 ;Branch if Z=0
59 FADD_code C 0044 0D22 MOV.W R2,R2 ;Branch if Z=0
60 FADD_code C 0046 4604 BNE LBL6 ;Branch if Z=0
61 FADD_code C 0048 706E BSET #6,R6L ;Bit set bit 6 of REL
62 FADD_code C 004A 721E BCLR #1,R6L ;Bit clear bit 1 of REL
63 FADD_code C 004C LBL6

232

RENESAS
FADD cod C 004C 777E  BLD  #7,R6L  ;Branch if not summand+addend=0
FADD cod C 004E 746E  BOR  #6,R6L
FADD cod C 0050 440C  BCC  LBL8
FADD cod C 0052 0331  ADD.W  R3,R1  ;Set summand and addend to result
FADD cod C 0054 0320  ADD.W  R3,R0
FADD cod C 0056 770E  BLD  #0,R6L
FADD cod C 0058 741E  BOR  #1,R6L
FADD cod C 005A 6770  BST  #7,R0H  ;Set sign bit
FADD cod C 005C 5470  RTS

; LBL8
FADD cod C 005E 7778  BLD  #7,R0L
FADD cod C 0060 1200  ROTXL R0H  ;Set exponent of summand to R0H
;
FADD cod C 0062 777A  BLD  #7,R2L
FADD cod C 0064 1202  ROTXL R2H  ;Set exponent of addend to R0L
;
FADD cod C 0066 7278  BCLR #7,R0L
FADD cod C 0068 0C00  MOV.B R0H,R0H
FADD cod C 006A 4704  BEQ LBL9  ;Branch if summand is normalized
FADD cod C 006C 7078  BSET #7,R0L  ;Set implicit MSB to summand
FADD cod C 006E 4002  BRA LBL10  ;Branch always

; LBL9
FADD cod C 0070 8001  ADD.B #H'01,R0H
FADD cod C 0072 8001  ADD.B #H'01,R0H
FADD cod C 0074 0C23  MOV.W R2,R3
FADD cod C 0076 4028  BRA LBL16  ;Branch always

; LBL10
FADD cod C 0078 727A  BCLR #7,R2L
FADD cod C 007A 0C22  MOV.B R2H,R2H
FADD cod C 007C 4704  BEQ LBL11  ;Branch if addend is normalized
FADD cod C 007E 707A  BSET #7,R2L  ;Set implicit MSB to addend
FADD cod C 0080 4002  BRA LBL12  ;Branch always

; LBL11
FADD cod C 0082 8201  ADD.B #H'01,R5H
FADD cod C 0084 4738  SUB.B R5L,R5H
FADD cod C 0086 451A  CMP.B #D'24,R5H  ;Set bit counter
FADD cod C 0088 450A  BCS LBL13  ;Branch if R5H<D'24
FADD cod C 008A 0C20  MOV.B R2H,R0H
FADD cod C 008C 79010000  MOV.W #H'0000,R1  ;Clear addend
FADD cod C 008E 0C98  MOV.B R1L,R0L
FADD cod C 0090 400C  BRA LBL16  ;Branch always

; LBL12
FADD cod C 0092 18D5  SUB.B R5L,R5H
FADD cod C 0094 4518  CMP.B R0L,R5H  ;Set bit counter
FADD cod C 0096 4508  BCS LBL13  ;Branch if R5H>R5L
FADD cod C 0098 70020000  MOV.W #H'0000,R2  ;Clear addend
FADD cod C 009A 0C23  MOV.B R1L,R0L
FADD cod C 009C 4028  BRA LBL16  ;Branch always

; LBL13
FADD cod C 009E 110A  SHLR R2L  ;Shift mantissa of addend 1 bit left
FADD cod C 00A0 1303  Roter R3H
FADD cod C 00A2 130D  Roter R3L
FADD cod C 00A4 1A05  DEC.B R5H  ;Decrement bit counter
FADD cod C 00A6 440F  SNE LBL13  ;Branch Z=0
FADD cod C 00A8 401C  BRA LBL16  ;Branch always

; LBL14
FADD cod C 00A0 185D  SHLR R5L  ;Shift mantissa of addend 1 bit right
FADD cod C 00A2 1438  Roter R1H
FADD cod C 00A4 1030  Roter R1L
FADD cod C 00A6 130B  DEC.B R5L  ;Decrement bit counter
FADD cod C 00A8 440F  SNE LBL15  ;Branch if Z=0
FADD cod C 00AA 400C  BRA LBL16  ;Branch always

; LBL15
FADD cod C 00A2 190D  SHLR R5L  ;Shift mantissa of addend 1 bit right
FADD cod C 00A4 413F  BCS LBL15  ;Branch if R5L>D'24
FADD cod C 00A6 402C  MOV.B R2H,R0H
FADD cod C 00A8 70010000  MOV.W #H'0000,R1  ;Clear summation
FADD cod C 00AA 0C28  MOV.B R1L,R0L
FADD cod C 00AC 400C  BRA LBL16  ;Branch always

; LBL16
FADD cod C 00A2 1503  SHLR R5L  ;Shift mantissa of addend 1 bit right
FADD cod C 00A4 1309  Roter R1L
FADD cod C 00A6 130B  DEC.B R5L  ;Decrement bit counter
FADD cod C 00A8 440F  SNE LBL15  ;Branch if Z=0
FADD cod C 00AA 400C  BRA LBL16  ;Branch always

; LBL17
FADD cod C 00A2 1503  SHLR R5L  ;Shift mantissa of addend 1 bit right
FADD cod C 00A4 1309  Roter R1L
FADD cod C 00A6 130B  DEC.B R5L  ;Decrement bit counter
FADD cod C 00A8 440F  SNE LBL15  ;Branch if Z=0
FADD cod C 00AA 400C  BRA LBL16  ;Branch always

; LBL18
134 FADD cod C 008E
135 FADD cod C 008E 7700
136 FADD cod C 00C0 751E
137 FADD cod C 00C2 4516
138                   ;Branch if different sign bit
139                   ;Addition mantissa
140 FADD cod C 00C4 0931
141 FADD cod C 00C6 0EA8
142 FADD cod C 00C8 442A
143 FADD cod C 00CA 1308
144 FADD cod C 00CE 1301
145 FADD cod C 00D0 8001
146 FADD cod C 00D2 A0FF
147 FADD cod C 00D4 4638
148 FADD cod C 00D6 5A000000
149                   ;Jump
150                   ;Substract mantissa
151 FADD cod C 00DA 1931
152 FADD cod C 00EC 1304
153 FADD cod C 00E0 F000
154 FADD cod C 00E2 5470
155 FADD cod C 00F0 9100
156 FADD cod C 00F2 9800
157 FADD cod C 00F4 440E
158 FADD cod C 00F6 1201
159 FADD cod C 00F8 1208
160 FADD cod C 00FA 1A00
161 FADD cod C 00FC 470C
162 FADD cod C 00F0 9100
163 FADD cod C 00F2 9800
164                   ;Clear RH
165                   ;Shift mantissa 1 bit left
166 FADD cod C 00F4 1009
167 FADD cod C 00F6 1208
168 FADD cod C 00FA 1A00
169 FADD cod C 00FC 470C
170 FADD cod C 00F0 9100
171 FADD cod C 00F2 9800
172 FADD cod C 00F4 44F4
173 FADD cod C 0100
174 FADD cod C 0100 0A00
175 FADD cod C 0102
176 FADD cod C 0104 1308
177 FADD cod C 0106 1300
178 FADD cod C 0108 1308
179 FADD cod C 010A 4004
180                   ;Branch always
181 FADD cod C 010A 45F4
182 FADD cod C 010C 40F4
183                   ;Change floating point format
184 FADD cod C 010E 1100
185 FADD cod C 010E 0100
186 FADD cod C 0110 6778
187 FADD cod C 0112 7700
188 FADD cod C 0114 6770
189 FADD cod C 0116 5470
190                   ;RTR
191                   ;END

*****TOTAL ERRORS    0
*****TOTAL WARNINGS  0
7.18 Multiplication of Short Floating-Point Numbers

MCU: H8/300 Series
     H8/300L Series

Label name: FMUL

7.18 Function

1. The software FMUL performs multiplication of short floating-point numbers placed in four general-purpose registers and places the result of multiplication in two of the four general-purpose registers.

2. All arguments used with the software FMUL are represented in short floating-point form.

7.18.1 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Multiplicand</td>
<td>R0, R1</td>
</tr>
<tr>
<td></td>
<td>Multiplier</td>
<td>R2, R3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R0, R1</td>
</tr>
<tr>
<td>Output</td>
<td>Result of multiplication</td>
<td></td>
</tr>
</tbody>
</table>

7.18.2 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>†</td>
<td>†</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>*</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

× : Unchanged
* : Indeterminate
† : Result
## 7.18.3 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>348</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>16</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>1078</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

## 7.18.4 Notes

The clock cycle count (16) in the specifications is for the example shown in figure 7.45.

For the format of floating-point numbers, see "About Short Floating-point Numbers <Reference>.”

## 7.18.5 Description

1. Details of functions
   a. The following arguments are used with the software FMUL:
      i. Input arguments:
         R0: Contains the upper 2 bytes of a short floating-point multiplicand.
         R1: Contains the lower 2 bytes of the short floating-point multiplicand.
         R2: Contains the upper 2 bytes of a short floating-point multiplier.
         R3: Contains the lower 2 bytes of the short floating-point multiplier.
(ii) Output arguments
R0: Contains the upper 2 bytes of the result.
R1: Contains the lower 2 bytes of the result.

b. Figure 7.45 shows an example of the software FMUL being executed. When the input arguments are set as shown in (a), the result of multiplication is placed in R0 and R1 as shown in (b).

<table>
<thead>
<tr>
<th>(1) Input arguments</th>
<th>R0, R1</th>
<th>8 0 0 0 0 0 1 0 1</th>
<th>R2, R3</th>
<th>7 F 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(H'80000001)</td>
<td>Sign bit=1, Exponent part=H'00, mantissa part=H'000001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(H'7F000000)</td>
<td>Sign bit=0, exponent part=H'6F, mantissa part=H'000000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| × \) | (2) Output arguments |
|---------------------| R0, R1 | B 4 8 0 0 0 0 0 0 |
| (H'B4800000) | Sign bit=1, exponent part=H'69, mantissa part=H'000000 |

**Figure 7.45  Example of Software FMUL Execution**

2. Notes on usage
a. The maximum and minimum values that can be handled by the software FADD are as follows:
   - Positive maximum H'7F800000
   - Positive minimum H'00000001
   - Negative maximum H'80000001
   - Negative minimum H'FF800000

b. All positive short floating-point numbers H'7F800000 to H'7FFFFFFF are treated as a maximum value (H'7F800000). All negative short floating-point numbers H'FF800000 to H'FFFFFFFF are treated as a minimum value (H'FFFFF800000).

c. As a maximum value is treated as infinity (∞), ∞ x 100 = ∞ or ∞ x (-100) = –∞. (See table 7.6)
Table 7.6 Examples of Operation with Maximum Values Used as Arguments

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Multiplier</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;H'7F800000</td>
<td>Positive number</td>
<td>H'7F800000 (+∞)</td>
</tr>
<tr>
<td>(∞)</td>
<td>Negative number</td>
<td>H'FF800000 (∞)</td>
</tr>
<tr>
<td>&lt;H'FF800000</td>
<td>Positive number</td>
<td>H'FF800000 (∞)</td>
</tr>
<tr>
<td>(∞)</td>
<td>Negative number</td>
<td>H'7F800000 (+∞)</td>
</tr>
<tr>
<td>Positive number</td>
<td>&gt;H'7F800000 (+∞)</td>
<td>H'7F800000 (+∞)</td>
</tr>
<tr>
<td></td>
<td>&lt;H'FF800000 (∞)</td>
<td>H'FF800000 (∞)</td>
</tr>
<tr>
<td>Negative number</td>
<td>&gt;H'7F800000 (+∞)</td>
<td>H'FF800000 (∞)</td>
</tr>
<tr>
<td></td>
<td>&lt;H'FF800000 (∞)</td>
<td>H'7F800000 (+∞)</td>
</tr>
</tbody>
</table>

d. H'80000000 is treated as H'00000000 (zero).
e. After execution of the software FMUL, the multiplicand and multiplier data are destroyed. If the input arguments are necessary after software FMUL execution, save them on memory.

3. Data memory
   The software FMUL does not use the data memory.
4. Example of use

Set a multiplicand and a multiplier in the general-purpose registers and call the software FMUL as a subroutine.

<table>
<thead>
<tr>
<th>WORK1</th>
<th>.RES. B 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORK2</td>
<td>.RES. B 2</td>
</tr>
<tr>
<td>WORK3</td>
<td>.RES. B 2</td>
</tr>
</tbody>
</table>

... Reserves a data memory area in which the user program places a multiplicand.

... Places a multiplier.

... the result of multiplication.

... MOV. W @WORK1,R0
MOV. W @WORK1+2,R1
MOV. W @WORK2,R2
MOV. W @WORK2+2,R3

... Places in R0 and R1 the multiplicand set by the user program.

... Places in R2 and R3 the multiplier set by the user program.

... Calls the software FMUL as a subroutine.

... MOV. W R0, @WORK3
MOV. W R1, @WORK3+2

... Places in R0 and R1 the result of multiplication set in the output argument.

5. Operation

Multiplication of short floating-point numbers is done in the following steps:

a. The software checks whether the multiplicand and multiplier are "0".
   (i) If either the multiplicand or multiplier is "0", H'00000000 is output.

b. The software checks whether the multiplicand and multiplier are infinite.
   If they are infinite, the values listed in table 7.6 are output.

c. Assume that the multiplicand is \( R_1 \) (sign bit=\( S_1 \), exponent part=\( \alpha_1 \), mantissa part=\( \beta_1 \)) and the multiplier is \( R_2 \) (sign bit=\( S_2 \), exponent part=\( \alpha_2 \), mantissa part=\( \beta_2 \)). Then \( R_1 \) and \( R_2 \) are given by

\[
R_1 = (-1)^{S_1} \times 2^{\alpha_1-127} \times \beta_1
\]

\[
R_2 = (-1)^{S_2} \times 2^{\alpha_2-127} \times \beta_2
\]

Multiplication of these two numbers is given by

\[
R_1 \times R_2 = (-1)^{S_1+S_2} \times 2^{\alpha_1+\alpha_2-254} \times \beta_1 \times \beta_2
\]

In the case of the floating-point format, the multiplication equation changes as follows, because H'7F (D'127) is added to the result of multiplication of the exponent parts:

\[
R_1 \times R_2 = (-1)^{S_1+S_2} \times 2^{\alpha_1+\alpha_2-254} \times \beta_1 \times \beta_2
\]

239
Thus, the multiplication is performed in the steps below:

(i) The software checks the sign bits of $R1 \times R2$.

(ii) Addition is done on the exponent parts.

   Both $\alpha_1$ and $\alpha_2$ involve addition of $H'7F$ (D'127) according to the floating-point format.

   As $H'7F$ (D'127) is also added to the result of multiplication, the operation goes as follows:

   $$(\alpha_1 - H'7F) + (\alpha_2 - H'7F) + H'7F = \alpha_1 + \alpha_2 - H'7F$$

   (In the case of the denormalized format, 1 is added to the exponent part before multiplication is done.)

(iii) Multiplication is done on the mantissa parts.

   This operation includes the value of the implicit MSB.

   (In the case of the denormalized format, the implicit MSB of the mantissa part is treated as "0".)

(iv) The result of multiplication is represented in floating-point format.
7.18.6 Flowchart

FMUL

#H'00 → R6L
#H'7F80 → R5

Bit 7 of R0H → C
C → Bit 0 of R6L

0 → Bit 7 of R0H

Bit 7 of R2H → C
C ⊕ Bit 0 of R6L → C
C → Bit 0 of R6L

0 → Bit 7 of R2H

R1 → R1

YES
R1 ≠ 0
NO
R0 → R0

YES
R0 ≠ 0
NO
R3 → R3

YES
R3 ≠ 0
NO
R2 → R2

YES
R2 ≠ 0
NO

LBL1

LBL2

Placed the sign bit of the multiplicand in bit 0 of R6L.

Clears bit 7 of R0H.

Processes the signs for multiplication and places the result in bit 0 of R6L.

Clears bit 7 of R2H.

Checks whether the multiplicand is "0"; branches to (1) if the multiplicand is "0" or to the next step if it is not "0".

Checks whether the multiplier is "0"; branches to (1) if the multiplier is "0" or to the next step if it is not "0".
Places a maximum positive number as output.

Branches if the multiplicand is infinite.

Goes to the next step if the multiplier is infinite.

Checks the sign for multiplication.

Places a maximum positive number as output.

Places a minimum negative number as output.
Bit 7 of R0L → C
Rotate R0H 1 bit left
R0H → R4L
#H'00 → R4H

Bit 7 of R2L → C
Rotate R2H 1 bit left
R2H → R5
#H'00 → R5H

0 → Bit 7 of R0L
R0H → R0H

YES
R0H = 0

NO
1 → Bit 7 of R0L

R+1 → R4

0 → Bit 7 of R2L
R2H → R2H

YES
R2H = 0

NO
1 → Bit 7 of R2L

R5+1 → R5

---

Places the exponent part of the multiplicand in R4.

---

Places the exponent part of the multiplier in R5.

---

Places "0" in bit 7 of R0L.

---

Branches if the exponent part of the multiplicand is "0".

---

Places "1" in bit 7 of R0L.

---

Increments R4.

---

Clears bit 7 of R2L.

---

Branches if the exponent part of the multiplier is "0".

---

Places "1" in bit 7 of R2L.

---

Increments R5.
ADD the exponent part of the multiplicand to that of the multiplier, and subtract H'7F from the result.

A

Push R4 and R6

R0 → R4
R1 → R5
R2L → R2H

MULA

Push R4 and R5

R3H → R2H

MULA

Push R4 and R5

R3L → R2H

MULA

R4 → R2
R5 → R3

B

C

Places R4 and R5 in R2 and R3, respectively.
245

Returns the result of (A).

Adds the result of (D) and that of (A).

Returns the result of (B).

Clears R1 to "0".

Adds the result of (C) to that of (B).

Returns the result of (A).

Add the result of (D) and that of (A).
Pop R6 and R4

R4 + #1 → R4

R4 → R4

R4 ≤ 0

R4 - #1 → R4

R4 → R4

R4 = 0

Shift R3L 1 bit left.
Rotate R3H, R2L, R2H, R1L, R1H 1 bit left

C = 0

Rotate R1H, R1L, R2H 1 bit right

R4 + #1 → R4

NO

YES

LBL12

NO

YES

LBL13

------ Returns the sign bit and exponent part.

------ Increments R4.

------ Increments R4.

------ Branches if R4 ≤ 0.

------ Decrement R4.

------ Branches if R4=0.

------ Shifts the mantissa part 1 bit to the left.

------ Branches if C=0.

------ Rotate the mantissa part 1 bit to the right.

------ Increments R4.
Branches if R4<'H'00FF.

Checks the sign bit.

Places maximum positive values in R0 and R1.

Places minimum negative values in R0 and R1.

Places "0" as output if R2H="0" and R1="0"; otherwise, branches.
LBL16

#H'0001 → R5
#D'24 → R6H

LBL17

Shift R1H 1 bit right
Rotate R1L and R2H 1 bit right

R4 + #1 → R4

R6H - #1 → R6H

R6H = 0
YES

NO

R4 = R5
YES

NO

LBL18

#H'0000 → R0
R0 → R1

RTS

Places "1" in R5. Places D'24 in R6H as maximum loop count.

Shifts the result of multiplication of the mantissa part 1 bit to the right.

Increments the exponent part.

Decrements R6H. Branches if R6H="0".

Branches if R4 is "1".

Places "0" as output.
Sets the exponent part to "0" if the MSB of the mantissa part is "0". (Denormalized format)

Sets the sign bit.

Places the exponent part in R0H and changes it to floating-point format.

Places the mantissa parts in respective registers.

Sets the exponent part to "0" if the MSB of the mantissa part is "0". (Denormalized format)

Changes the exponent part to floating-point format.

Sets the sign bit.
Places the mantissa part of the multiplicand.

Finds partial products of R2H x the mantissa part of the multiplicand and adds the results.
7.18.7  Program List

*** H8/300 ASSEMBLER                VER 1.0B **   08/18/92 10:22:23

PROGRAM NAME =

1 ;********************************************************************
2 ;*
3 ;* 00 - NAME :FLOATING POINT MULTIPLICATION (FMUL)
4 ;*
5 ;********************************************************************
6 ;*
7 ; ENTRY :R0 (UPPER WORD OF MULTIPLICAND)
8 ; R1 (LOWER WORD OF MULTIPLICAND)
9 ; R2 (UPPER WORD OF MULTIPLIER)
10 ; R3 (LOWER WORD OF MULTIPLIER)
11 ;*
12 ; RETURNS :R0 (UPPER WORD OF RESULT)
13 ; R1 (LOWER WORD OF RESULT)
14 ;*
15 ;********************************************************************

16 ;
17 FMUL_cod C 0000
18 .SECTION FMUL_code,CODE,ALIGN=2
19 ;
20 FMUL_cod C 00000000 FMUL.equ $ ;Entry point
21 FMUL_cod C 0000 0000 MOV.B R0,REL ;Clear REL
22 FMUL_cod C 0002 7805 MOV.W R0,R5 ;Set "H'7F80"
23 ;
24 FMUL_cod C 0006 7770 SLD #7,R0H ;Set sign bit of multiplicand
25 FMUL_cod C 0008 770E BST #0,R6L ; to bit 0 of REL
26 FMUL_cod C 000A 7270 BCLR #7,R0H ;Bit clear bit 7 of R0H
27 ;
28 FMUL_cod C 000C 7772 SLD #7,R2H ;
29 FMUL_cod C 000E 750E MULR #R0,R6L ;Set sign bit of result
30 FMUL_cod C 0010 470E BST #0,R6L ; to bit 0 of REL
31 FMUL_cod C 0012 7272 BCLR #7,R2H ;Bit clear bit 7 of R2H
32 ;
33 FMUL_cod C 0014 0011 MOV.W R1,R1
34 FMUL_cod C 0016 4404 BNE LBL1
35 FMUL_cod C 0018 0D00 MOV.W R0,R0 ;Branch if R1=R0=0
36 FMUL_cod C 001A 4708 BEQ LBL2 ;Branch if R1=R0=0
37 FMUL_cod C 001C LBL1
38 FMUL_cod C 001C 0D33 MOV.W R3,R3
39 FMUL_cod C 001E 440C BNE LBL5 ;Branch if not R3=0
40 FMUL_cod C 0020 0022 MOV.W R2,R2
41 FMUL_cod C 0022 4408 BNE LBL5 ;Branch if not R2=0
42 ;
43 FMUL_cod C 0024 LBL2
44 FMUL_cod C 0024 79000000 MOV.W #H'0000,R0 ;Set 0 to result
45 FMUL_cod C 0028 0001 MOV.W R0,R1
46 FMUL_cod C 002A 8470 RTS
47 ;
48 FMUL_cod C 002C LBL3
49 FMUL_cod C 002C 10D5 CMP.W R0,R5 ;Branch if R0<>R5
50 FMUL_cod C 002E 4304 BLS LBL4 ;Branch if R0>=R5
51 FMUL_cod C 0030 10D3 CMP.W R2,R5 ;Branch if R2<>R5
52 FMUL_cod C 0032 4218 BHI LBL4 ;Branch if R2>=R5
53 FMUL_cod C 0034 LBL4
54 FMUL_cod C 0034 770E SLD #0,R6L ;Load sign bit
55 FMUL_cod C 0036 450A BCS LBL6 ;Branch if C=1
56 FMUL_cod C 0038 LBL5
57 FMUL_cod C 0038 78000000 MOV.W #H'FF80,R0 ;Set #H'FF800000 to result
58 FMUL_cod C 003C 70010000 MOV.W #H'0000,R1
59 FMUL_cod C 0040 5470 RTS
60 FMUL_cod C 0042 LBL6
61 FMUL_cod C 0042 78000000 MOV.W #H'FF80,R0 ;Set #H'FF800000 to result
62 FMUL_cod C 0046 70010000 MOV.W #H'0000,R1
63 FMUL_cod C 004A 5470 RTS
252
253

134
135 FMUL_cod C 00BC 4D76
136 FMUL_cod C 00BE 4D74
137 FMUL_cod C 00C0 0B04
138 FMUL_cod C 00C2 0B04
139
140 FMUL_cod C 00C4 474A
141 FMUL_cod C 00C6 4848
142 FMUL_cod C 00C8
143 FMUL_cod C 00CB 1B04
144 FMUL_cod C 00C9 1B04
145 FMUL_cod C 00CA 0D01
146 FMUL_cod C 00CB 1A06
147 FMUL_cod C 00CD 0B04
148 FMUL_cod C 00CE 0B04
149 FMUL_cod C 00CF 0B04
150 FMUL_cod C 00D0 0B04
151 FMUL_cod C 00D1 0B04
152 FMUL_cod C 00D2 0B04
153 FMUL_cod C 00D3 0B04
154 FMUL_cod C 00D4 0B04
155 FMUL_cod C 00D5 0B04
156 FMUL_cod C 00D6 0B04
157 FMUL_cod C 00D7 0B04
158
159 FMUL_cod C 00D8 79050000
160 FMUL_cod C 00D9 79050000
161 FMUL_cod C 00DA 79050000
162 FMUL_cod C 00DB 79050000
163 FMUL_cod C 00DC 79050000
164 FMUL_cod C 00DD 79050000
165 FMUL_cod C 00DE 79050000
166 FMUL_cod C 00DF 79050000
167
168 FMUL_cod C 00E0
169 FMUL_cod C 00E1
170 FMUL_cod C 00E2
171 FMUL_cod C 00E3
172
173 FMUL_cod C 00E4
174 FMUL_cod C 00E5
175 FMUL_cod C 00E6
176 FMUL_cod C 00E7
177 FMUL_cod C 00E8
178 FMUL_cod C 00E9
179 FMUL_cod C 00EA
180
181 FMUL_cod C 00EB
182 FMUL_cod C 00EC
183 FMUL_cod C 00ED
184 FMUL_cod C 00EE
185 FMUL_cod C 00EF
186 FMUL_cod C 00F0
187 FMUL_cod C 00F1
188 FMUL_cod C 00F2
189 FMUL_cod C 00F3
190 FMUL_cod C 00F4
191 FMUL_cod C 00F5
192 FMUL_cod C 00F6
193 FMUL_cod C 00F7
194 FMUL_cod C 00F8
195 FMUL_cod C 00F9
196 FMUL_cod C 00FA
197 FMUL_cod C 00FB
198
199 FMUL_cod C 0100
200 FMUL_cod C 0101
201 FMUL_cod C 0102
202 FMUL_cod C 0103
203
204 FMUL_cod C 013E 00C0  MOV.B #R4L,RSH
205 FMUL_cod C 013E 7778  BLD #T,RSL
206 FMUL_cod C 013A 4502  BCS LBL20 ;Branch if C=1
207 FMUL_cod C 013C F000  MOV.B #H'00,RSH
208 FMUL_cod C 013E LBL20 ;Change floating point format
209 FMUL_cod C 013E 1100  SRLA RSH
210 FMUL_cod C 0140 6778  BLD #7,R0L
211 FMUL_cod C 0142 770E  BLD #R3L
212 FMUL_cod C 0144 4177  BRT #7,RSH
213 FMUL_cod C 0148 5470  RTS
214 ;
215 ;-----------------------------------------------
216 ;
217 FMUL_cod C 0148  MULA ;R2H * (R0L:R1) -> (R4:R5)
218 FMUL_cod C 0148 0D04  MOV.W R0,R4 ;R0  -> R4
219 FMUL_cod C 014A 0D15  MOV.W R1,R5 ;R1  -> R5
220 FMUL_cod C 014C 0C1E  MOV.B R1H,R6L ;R1H -> R6L
221 ;
222 FMUL_cod C 014E 5025  MULW R2H,R5 ;R2H * R5L -> R5
223 FMUL_cod C 0150 5026  MULW R2H,R6 ;R2H * R6L -> R6
224 FMUL_cod C 0152 5024  MULW R2H,R4 ;R2H * R4L -> R4
225 ;
226 FMUL_cod C 0154 08E5  ADD.W R6L,R5H ;R5H + R6L -> R5
227 FMUL_cod C 0156 08E6  ADD.W R4H,R6L ;R4L + R6H + C -> R4L
228 FMUL_cod C 0158 9400  ADD.W #H'00,R4H ;R4H + #H'00 + C -> R4H

WASHINGTON ASSEMBLER              VER 1.0B **   08/18/92 10:22:23                                                      PAGE
*** H8/300 ASSEMBLER                VER 1.0B **   08/18/92 10:22:23                                                      PAGE

PROGRAM NAME =

229 FMUL_cod C 013E 5470  RTS
230 ;
231 .END

*****TOTAL ERRORS       0
*****TOTAL WARNINGS     0
## 7.19 Square Root of a 32-Bit Binary Number

**MCU:** H8/300 Series  
H8/300L Series  

**Label name:** SQRT

### 7.19.1 Function

1. The software SQRT finds the square root of a 32-bit binary number and outputs the result in 16-bit binary format.
2. All arguments used with the software SQRT are represented in unsigned integers.
3. All data is manipulated on general-purpose registers.

### 7.19.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input 32-bit binary number</td>
<td>R4, R5</td>
<td>4</td>
</tr>
<tr>
<td>Output Square root</td>
<td>R3</td>
<td>2</td>
</tr>
</tbody>
</table>

### 7.19.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>†</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

× : Unchanged  
*: Indeterminate  
†: Result
7.19.4 Specifications

<table>
<thead>
<tr>
<th>Program memory (bytes)</th>
<th>94</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>1340</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

7.19.5 Notes

The clock cycle count (1340) in the specifications is for the example shown in figure 7.46.

7.19.6 Description

1. Details of functions
   a. The following arguments are used with the software SQRT:
      R4: Contains, as an input argument, the upper word of a 32-bit binary number whose square root is to be found.
      R5: Contains, as an input argument, the lower word of the 32-bit binary number whose square root is to be found.
      R3: Contains, as an output argument, the square root of the 32-bit binary number.
   b. Figure 7.46 shows an example of the software SQRT being executed. When the input arguments are set as shown in (1), the square root is placed in R3 shown in (2).
2. Notes on usage
   a. When upper bits are not used (see figure 7.47), set 0's in them; otherwise, no correct result can be obtained because the square root is found on numbers including indeterminate data placed in the upper bits.

3. Data memory
   The software SQRT does not use the data memory.
4. Example of use

Set a 32-bit decimal number whose square root is to be found and call the software SQRT as a subroutine.

```
WORK1 . RES. W 2           Reserves a data memory area in which the user program places a 32-bit binary number whose square root is to be found.
WORK2 . RES. W 2           Reserves a data memory area in which the user program places the square root (16-bit binary) of the 32-bit binary number.
  MOV. W @WORK1, R4        Places in the input argument the 32-bit binary number set by the user program.
  MOV. W @WORK1+2, R5      Places the 16-bit binary square root (set in the output argument) in the data memory area of the user program.
  JSR @SQRT                Calls the software SQRT as a subroutine.
```

5. Operation

a. Figure 7.48 shows the method of finding the square root H'05 (binary) of H'22 (a 16-bit binary).

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.48 Computation to Find Square Root
```

(i) As shown in figure 7.48, the square root of a binary number can be found by processing the number by 2 bits in bit-descending order.
(ii) The square root \((1)\) is equal to \(É\) divided by 2. The software SQRT computes \(É\) to find the square root.

b. The program is executed in the following steps:

(i) The number of steps \((D'16)\) in which the 32-bit binary number is processed by 2 bits is placed in R6L.

(ii) The square root areas R2 and R3 and the work areas R0 and R1 are cleared.

(iii) R4, R5 and R0, R1 are rotated 2 bits to the left to place the upper 2 bits of the input square root in R0 and R1.

(iv) "1" is set in R2 and R3. (2)

(v) R2 and R3 are subtracted from R0 and R1 to find the difference. (D, (2), (3), (4)) The difference is placed in R0 and R1.

(vi) If the result is positive, R2 and R3 are incremented. (A, -(4))

If the result is negative, R2 and R3 are decremented, and R2 and R3 are added to R0 and R1. (D, E, (6))

c. In the software SQRT, R6 is decremented each time the process (iii) through (vi) of (b) is done. This processing continued until R6 reaches "0".
Shift the 32-bit binary number 2 bits to the left to take out the value of the upper 2 bits.

Rotates the work areas 2 bit to the left and places the result in the lower 2 bits.

Clears the work areas (R0 and R1) and the result areas (R2 and R3).

Loads D'16 (the number of processing steps) to the loop counter (R6L).

Shift R5L 1 bit left
Rotate R5H 1 bit left
Rotate R4L 1 bit left
Rotate R4H 1 bit left
Rotate R1L 1 bit left
Rotate R1H 1 bit left
Rotate R0L 1 bit left
Rotate R0H 1 bit left

Shift the 32-bit binary number 2 bits to the left to take out the value of the upper 2 bits. Rotates the work areas 2 bit to the left and places the result in the lower 2 bits.
Subtracts the value of the result areas from the value of the work areas and places the result in the work areas. Branches if no borrow occurs.

Rotates the result areas 1 bit to the left and places "1" in the MSB.

Adds the value of the work areas to the value of the result areas and returns the value of the work areas.

Clears the MSB ("1") of the result areas.
Branches 16 times.

Adds "1" to the value of the result areas.

The value of the result areas is divided by 2 to obtain the square root.
### 7.19.8  Program List

```
*** 88/100 ASSEMBLER                VER 1.0B **   08/18/92 10:23:40

PROGRAM NAME = 32 BIT SQUARE ROOT (SQRT)

1 ;********************************************************************
2 ;* 00 - NAME : 32 BIT SQUARE ROOT (SQRT)
3 ;* ENTRY :R4,R5 (32 BIT BINARY)
4 ;* RETURN :R3 (SQUARE ROOT)
5 ;********************************************************************
6 ;*
7 ;* SQRT_code C 0000 .SECTION SQRT_code,CODE,ALIGN=2
8 ;*
9 ;* LBL1
10 ;*
11 ;*
12 ;*
13 SQRT_cod C 0000 .EXPORT SQRT
14 ;*
15 ;*
16 SQRT_cod C 00000000 .SECTION SQRT_code,CODE,ALIGN=2
17 SQRT_cod C 00000000 MOV.B #D'16,R6L ;Set shift counter
18 SQRT_cod C 00000000 MOV.W #H'0000,R0 ;Clear R0
19 SQRT_cod C 00000000 MOV.W R0,R1 ;Clear R1
20 SQRT_cod C 00000000 MOV.W R0,R2 ;Clear R2
21 SQRT_cod C 00000000 MOV.W R0,R3 ;Clear R3
22 SQRT_cod C 000C .SECTION SQRT_code,CODE,ALIGN=2
23 SQRT_cod C 000C .SECTION SQRT_code,CODE,ALIGN=2
24 .SECTION SQRT_code,CODE,ALIGN=2
25 SQRT_code C 0000 SQRT .EQU $ ;Entry point
26 SQRT_code C 0000 SQRT .EQU $ ;Entry point
27 SQRT_code C 0000 SQRT .EQU $ ;Entry point
28 SQRT_code C 0000 SQRT .EQU $ ;Entry point
29 SQRT_code C 0000 SQRT .EQU $ ;Entry point
30 SQRT_code C 0000 SQRT .EQU $ ;Entry point
31 SQRT_code C 0000 SQRT .EQU $ ;Entry point
32 SQRT_code C 0000 SQRT .EQU $ ;Entry point
33 SQRT_code C 0000 SQRT .EQU $ ;Entry point
34 SQRT_code C 0000 SQRT .EQU $ ;Entry point
35 SQRT_code C 0000 SQRT .EQU $ ;Entry point
36 SQRT_code C 0000 SQRT .EQU $ ;Entry point
37 SQRT_code C 0000 SQRT .EQU $ ;Entry point
38 SQRT_code C 0000 SQRT .EQU $ ;Entry point
39 SQRT_code C 0000 SQRT .EQU $ ;Entry point
40 SQRT_code C 0000 SQRT .EQU $ ;Entry point
41 SQRT_code C 0000 SQRT .EQU $ ;Entry point
42 SQRT_code C 0000 SQRT .EQU $ ;Entry point
43 SQRT_code C 0000 SQRT .EQU $ ;Entry point
44 SQRT_code C 0000 SQRT .EQU $ ;Entry point
45 SQRT_code C 0000 SQRT .EQU $ ;Entry point
46 SQRT_code C 0000 SQRT .EQU $ ;Entry point
47 SQRT_code C 0000 SQRT .EQU $ ;Entry point
48 SQRT_code C 0000 SQRT .EQU $ ;Entry point
49 SQRT_code C 0000 SQRT .EQU $ ;Entry point
50 SQRT_code C 0000 SQRT .EQU $ ;Entry point
51 SQRT_code C 0000 SQRT .EQU $ ;Entry point
52 SQRT_code C 0000 SQRT .EQU $ ;Entry point
53 SQRT_code C 0000 SQRT .EQU $ ;Entry point
54 SQRT_code C 0000 SQRT .EQU $ ;Entry point
55 SQRT_code C 0000 SQRT .EQU $ ;Entry point
56 SQRT_code C 0000 SQRT .EQU $ ;Entry point
57 SQRT_code C 0000 SQRT .EQU $ ;Entry point
58 SQRT_code C 0000 SQRT .EQU $ ;Entry point
59 SQRT_code C 0000 SQRT .EQU $ ;Entry point
60 SQRT_code C 0000 SQRT .EQU $ ;Entry point
61 SQRT_code C 0000 SQRT .EQU $ ;Entry point
62 SQRT_code C 0000 SQRT .EQU $ ;Entry point
63 SQRT_code C 0000 SQRT .EQU $ ;Entry point
```

---

**Note:** The above code listing is for a 32-bit square root algorithm implemented in H8/300 Assembler. It includes initialization of shift counter, setting up necessary registers, and the main loop for calculating the square root through bit shifting and arithmetic operations.
64 SQRT cod C 0058 1303       ROTXR.B R3H       ;Rotate square root
65 SQRT cod C 005A 130B       ROTXR.B R3L
66 SQRT cod C 005C 5470       RTS
67
68
*****TOTAL ERRORS 0
*****TOTAL WARNINGS 0
Section 8  DECIMAL ↔ HEXADECIMAL CHANGE

8.1  Change a 2-Byte Hexadecimal Number to a 5-Character BCD Number

MCU:  H8/300 Series
H8/300L Series

Label name:  HEX

8.1.1  Function

1. The software HEX changes a 2-byte hexadecimal number (placed in a general-purpose register) to a 5-character BCD (binary-coded decimal) number and places the result of change in general-purpose registers.
2. All arguments used with the software HEX are represented in unsigned integers.
3. All data is manipulated on general-purpose registers.

8.1.2  Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>2-byte hexadecimal number</td>
<td>R0</td>
</tr>
<tr>
<td>Output</td>
<td>5-character BCD number (upper 1 character)</td>
<td>R2L</td>
</tr>
<tr>
<td></td>
<td>5-character BCD number (lower 4 characters)</td>
<td>R3</td>
</tr>
</tbody>
</table>

8.1.3  Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2H</th>
<th>R2L</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>•</td>
<td>×</td>
<td>†</td>
<td>†</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>•</td>
<td>×</td>
<td>•</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

× : Unchanged
• : Indeterminate
† : Result
8.1.4 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>30</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>368</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

8.1.5 Description

1. Details of functions
   a. The following arguments are used with the software HEX:
      R0: Contains a 2-byte hexadecimal number as an input argument.
      R2L: Contains the upper 1 character (1 byte) of a 5-character BCD number as an output argument.
      R3: Contains the lower 4 characters (2 bytes) of the 5-character BCD number as an output argument.

Figure 8.1 shows the formats of the input and output arguments.
b. Figure 8.2 shows an example of the software HEX being executed. When the input argument is set as shown in (1), the 5-character BCD number is placed in R2L and R3 as shown in (2).

2. Notes on usage
When upper bits are not used (see figure 8.3), set 0’s in them; otherwise, no correct result can be obtained because computation is made on numbers including indeterminate data placed in the upper bits.

3. Data memory
The software HEX does not use the data memory.
4. Example of use
Set a 2-byte hexadecimal number in R0 and call the software HEX as a subroutine.

<table>
<thead>
<tr>
<th>WORK1</th>
<th>. RES. W 1</th>
<th>Reserves a data memory area in which the user program places a 2-byte hexadecimal number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORK2</td>
<td>. RES. B 3</td>
<td>Reserves a data memory area in which the user program places a 5-character BCD number (3 bytes).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Places in the input argument the 2-byte hexadecimal number set by the user program.</td>
</tr>
<tr>
<td>MOV. W</td>
<td>@WORK1, R0</td>
<td>Calls the software HEX as a subroutine.</td>
</tr>
<tr>
<td>JSR</td>
<td>@HEX</td>
<td></td>
</tr>
<tr>
<td>MOV. B</td>
<td>R2L,@WORK2</td>
<td></td>
</tr>
<tr>
<td>MOV. B</td>
<td>R3H,@WORK2+1</td>
<td></td>
</tr>
<tr>
<td>MOV. B</td>
<td>R3L,@WORK2+2</td>
<td></td>
</tr>
</tbody>
</table>

5. Operation
a. A 4-bit binary number "B3B2B1B0" is represented by equations 1 and 2 below:

\[
B_3B_2B_1B_0 = B_3 \times 2^3 + B_2 \times 2^2 + B_1 \times 2^1 + B_0 \times 2^0 \quad \text{(equation 1)}
\]

\[
= ( ( B_3 \times 2 + B_2 ) \times 2 + B_1 ) \times 2 + B_0 \quad \text{(equation 2)}
\]

\[
\begin{array}{c}
\alpha \\
\beta \\
\gamma
\end{array}
\]

b. First, equation 2 is used to compute \( \alpha = B_3 \times 2 + B_2 \) (see figure 8.4) by executing an add instruction (the ADD.B instruction) and decimal correction (the DAA instruction). Next, a series of arithmetic operations such as \( \beta = \alpha \times 2 + B_1 \) and \( \gamma = \beta \times 2 + B_0 \) are performed to find a 5-character BCD number as the result.

c. The software HEX uses R0 (input) and R2L and R3 (outputs) to compute \( \alpha = B_3 \times 2 + B_2 \).

(i) R2H is used as the counter that shifts R0 (containing the input argument) bit by bit. D'16 is set in R2H for a total of 16 shifts.
(ii) R0 (containing the 2-byte hexadecimal number) is shifted 1 bit to the left, and the most significant bit is placed in the C flag.

(iii) R2L and R3 (containing the 5-character BCD number) are processed in ascending order, as follows:

\[
\begin{align*}
R3L + R3L + C & \to R3L \quad \text{Decimal correction of } R3L \\
R3H + R3H + C & \to R3H \quad \text{Decimal correction of } R3H \\
R2L + R2L + C & \to R2L \quad \text{Decimal correction of } R2L
\end{align*}
\]

Thus, \( \alpha = B_3 \times 2 + B_2 \) has been computed.

(iv) In the software HEX, R2H is decremented each time the process (ii) to (iii) is performed. This processing continues until R2H reaches "0".
8.1.6 Flowchart

- HEX
  - #H'0000 → R2
  - R2 → R3
  - #D'16 → R2H
    - Clears the work area (to contain the result)
    - Places the loop count in R2H.
    - Places the MSB of a 2-byte hexadecimal number in the C bit.
    - Makes the value of work areas (R2L and R3) equivalent to doubled BCD and adds the MSB of 2-byte hexadecimal number (R0).
    - Repeats 16 times.

- LOOP
- Shift R0L 1 bit left
- Rotate R0H 1 bit left
- R3L + R3L + C → R3L
  - Decimal correction of R3L
- R3H + R3H + C → R3H
  - Decimal correction of R3H
- R2L + R2L + C → R2L
  - Decimal correction of R2L
- R2H – #1 → R2H
- YES
- R2H ≠ 0
- RTS
- NO
8.1.7 Program List

*** H8/300 ASSEMBLER ** VER 1.0B ** 08/18/92 10:24:23

PROGRAM NAME =

1 ;********************************************************************
2 ;* 00 - NAME :CHANGE 2 BYTE HEXADECIMAL TO BCD (HEX)
3 ;*
4 ;********************************************************************
5 ;*
6 ; ENTRY :R0 (HEXADECIMAL)
7 ;*
8 ; RETURNS :R2L (UPPER 1 CHARACTER (BY BCD))
9 ; R3 (LOWER 4 CHARACTER (BY BCD))
10 ;*
11 ;********************************************************************
12 ;*
13 ;********************************************************************
14 ;
15 HEX_code C 0000 .SECTION HEX_code,CODE,ALIGN=2
16 .EXPORT  HEX
17 ;
18 HEX_code C 00000000 HEX  EQU 5 ;Entry point
19 HEX_code C 00005902000 MOV.W R7'0000,R2 ;Clear R2
20 HEX_code C 0004 0023 MOV.W R2,R3 ;Clear R3
21 HEX_code C 0006 F210 MOV.B R5'16,R2H ;Set bit counter
22 HEX_code C 0008 LOOP SHLL.B R0L ;Shift hexadecimal 1 bit left
23 HEX_code C 0008 1000 BXHL.B R0H
24 HEX_code C 000A 1200
25 ;
26 HEX_code C 000C 02BB ADDX.B R3L,R3L ;R3L + R3L --> R3L
27 HEX_code C 000E 070B CAA R3L ;Decimal adjust R3L
28 HEX_code C 0010 0233 ADDX.B R3H,R3H ;R3H + R3H + C --> R3H
29 HEX_code C 0012 0033 CAA R3H ;Decimal adjust R3H
30 HEX_code C 0014 002A ADDX.B R2L,R2L ;R2L + R2L + C --> R2L
31 HEX_code C 0016 0020 CAA R2L ;Decimal adjust R2L
32 ;
33 HEX_code C 0018 1AE2 DEC.B R2H ;Decrement R2H
34 HEX_code C 001A A4EC SNE LOOP ;Branch Z=0
35 HEX_code C 001C 5470 RTS
36 ;
37 ;END

*****TOTAL ERRORS 0
*****TOTAL WARNINGS 0
8.2 Change a 5-Character BCD Number to a 2-Byte Hexadecimal Number

MCU: H8/300 Series
H8/300L Series

Label name: BCD

8.2.1 Function

1. The software BCD changes a 5-character BCD (binary-coded decimal) Number (3 bytes, placed in a general-purpose registers) to a 2-byte hexadecimal number and places the result of change in a general-purpose register.
2. All data is manipulated on general-purpose registers.
3. The 5-character BCD number can be up to H'65535.

8.2.2 Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>R0L</td>
<td>1</td>
</tr>
<tr>
<td>5-character BCD number (upper 1 character)</td>
<td>R0L</td>
<td>1</td>
</tr>
<tr>
<td>5-character BCD number (lower 4 characters)</td>
<td>R1</td>
<td>2</td>
</tr>
<tr>
<td>Output</td>
<td>R2</td>
<td>2</td>
</tr>
<tr>
<td>2-byte hexadecimal number</td>
<td>R2</td>
<td>2</td>
</tr>
</tbody>
</table>

8.2.3 Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0H</th>
<th>R0L</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5H</th>
<th>R5L</th>
<th>R6</th>
<th>R7</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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

× : Unchanged
• : Indeterminate
‡ : Result
### 8.2.4 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>64</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>2</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>210</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

### 8.2.5 Description

1. Details of functions

   a. The following arguments are used with the software BCD:
      
      - **R0L**: Contains the upper 1 character (1 byte) of a 5-character BCD number as an input argument.
      - **R1**: Contains the lower 4 characters (2 bytes) of the 5-character BCD number as an input argument.
      - **R2**: Contains a 2-byte hexadecimal number as an output argument.

      Figure 8.5 shows the formats of the input and output arguments.
b. Figure 8.6 shows an example of the software BCD being executed. When the input argument is set as shown in (1), the 2-byte hexadecimal number is placed in R2 as shown in (2).

2. Notes on usage
   a. The values of bits 4 through 7 of R0L (containing the upper 1 character of the 5-character BCD number) remain unchanged. They are cleared to “0” after execution of the software BCD.
   b. The 5-character BCD number can be up to H'65535.
   c. When upper bits are not used, set 0's in them; otherwise, no correct result can be obtained because computation is made on numbers including indeterminate data placed in the upper bits.

3. Data memory
   The software BCD does not use the data memory.

4. Example of use
   Set a 5-character BCD number in the input arguments and call the software BCD as a subroutine.
WORK1 . RES B 3 Reserves a data memory area in which the user program places a 5-character BCD number (3 bytes).

WORK2 . RES B 2 Reserves a data memory area in which the user program places a 2-byte hexadecimal number.

MOV. B @WORK1, R0L Places in the input argument the 5-character BCD number set by the user program.
MOV. B @WORK1+1, R1H
MOV. B @WORK1+2, R1L

MOV. B @WORK2, R2H Places the 2-byte hexadecimal number (set in the output argument) in the data memory area of the user program.
MOV. B @WORK2+1, R2L

JSR @BCD Calls the software BCD as a subroutine.

5. Operation
a. The software BCD consists of two processes:
   (i) Popping up the 5-character BCD number character by character
   (ii) Changing the popped-up data to a hexadecimal number on a 4-bit basis.
b. Figure 8.7 shows the method of computing a 1-character (4-bit) number.

Figure 8.7 Method of Dividing 1-Byte Register Data by 2

(i) $\text{H'}04$ is placed for computation of the 5 characters.
(ii) The 5-character BCD number ($R0L$, $R1H$, $R1L$) is transferred to $R6L$ starting with the most significant byte. Then the upper or lower 4 bits are selected.
(iii) $R0H$ is decremented each time the process (ii) is performed.
(iv) When the process (ii) is performed, the software checks whether the counter ($R0H$) is even or odd.
   - When $R0H$ is odd, $R6L$ is ANDed with $\text{H'}0F$ to pop up the lower 4 bits.
• When ROH is even, R6L is shifted 4 bits to the right to pop up the upper 4 bits.

c. The BCD number is changed to a hexadecimal number in the following steps:

(i) A 4-character BCD "D_3 D_2 D_1 D_0" is represented by equations 1 and 2 below:

\[
D_3 \times 10^3 + D_2 \times 10^2 + D_1 \times 10^1 + D_0 \times 10^0 = (D_3 \times 10 + D_2) \times 10 + D_1 \times 10 + D_0.
\]  

Figure 8.8 4-character BCD Number "D_3 D_2 D_1 D_0"

(ii) First, equation 2 is used to compute \( \alpha = D_3 \times 10 + D_2 \) (see figure 8.8). Next, a series of arithmetic operations such as \( \beta = \alpha \times 10 + D_1 \) and \( \gamma = \beta \times 10 + D_0 \) are performed to find a hexadecimal number as the result.

(iii) Equations 3 and 4 are used to compute \( D_3 \times 10 \):

\[
D_3 \times 10 = D_3 \times (2 + 8) \quad \text{......... (equation 3)}
\]

\[
= D_3 \times 2 \times (1 + 2^2) \quad \text{......... (equation 4)}
\]

(iv) The software HEX uses R2 and R3 to compute equation 4 by taking the following steps:

1. Places \( D_3 \) in R2 and shifts it 1 bit to the left.
2. Transfers R2 to R3 and shifts it 1 bit to the left.
3. Adds R3 to R2.

d. The hexadecimal form of the 2-byte BCD number can be obtained by repeating the process b. to c. five times.
8.2.6 Flowchart

- BCD
- \#H'04 \rightarrow R0H
- \#H'0000 \rightarrow R2
  \hspace{0.5cm} R2 \rightarrow R6
- R2L + R0L \rightarrow R2L
- R1H \rightarrow R6L

\{ Places H'04 in the counter (R0H). \}
\{ Clears R2 and R6. \}
\{ Adds the MSB of the 5-character BCD number (R0L) to R2L. \}

\{ Transfers the lower 4 characters of the BCD number (R1) byte by byte to R6L and calls the subroutine TRANS. The subroutine computes the number character by character to obtain a 2-byte hexadecimal number as the result. \}

- TRANS
- R1L \rightarrow R6L

RTS
TRANS

R6L → R5L

Bit 0 of R0H → C flag

C = 0

YES

LBL1

R5L → R6L

R6L + #H'0F → R6L

LBL2

Shift R6L 4 bits right

LBL3

Shift R2 1 bit left

R2 → R3

Shift R2 2 bits left

R2 + R3 → R2

R6L + R2L → R2L

R2H + #H'00 + C → R2H

R0H - #1 → R0H

Bit 0 of R0H → C flag

C = 1

NO

RTS

NO

YES

Saves R6L to R5L.

Loads bit 0 of R0H to the C flag. Branches if C=0 (R0H: even).

Returns the saved value to R6L.

Pops up the lower 4 bits of R6L.

Pops up the upper 4 bits of R6L.

Multiplies R2 by 10 through shifts and additions and adds 1 character of the BCD to R2 to obtain a hexadecimal number.

Adds the lower characters.

Decrements R0H.

Loading bit 0 of R0H to the C flag. Branches if C=1 (R0H: odd).
### 8.2.7 Program List

```
*** H8/300 ASSEMBLER                VER 1.0B **   08/22/92 11:09:49

PROGRAM NAME =
1                                      ;********************************************************************
2                                      ;*
3                                      ;* 00 - NAME :CHANGE 5 CHARACTER
4                                      ;*   TO 2 BYTE HEXADECIMAL (BCD)
5                                      ;*
6                                      ;********************************************************************
7                                      ;*
8                                      ;* ENTRY :R0L (UPPER 1 CHAR (BY BCD))
9                                      ;*  R1 (LOWER 4 CHAR (BY BCD))
10                                     ;*
11                                      ;* RETURN :R2 (2 BYTE HEXADECIMAL)
12                                      ;*
13                                      ;********************************************************************
14                                      ;
15 BCD_code C 0000 .SECTION BCD_code,CODE,ALIGN=2
16                                      ; .EXPORT BCD
17                                      ; ;--------------------------------------------------------------
18 BCD_code C 00000000 BCD .EQU 5 ;Entry point
19 BCD_code C 0000 F004 MOV.B #H'04,R0H ;Set bit counter
20 BCD_code C 0002 70020000 MOV.W #H'0000,R2 ;Clear R2
21 BCD_code C 0006 0006 MOV.W R2,R6 ;Clear R6
22                                      ;
23 BCD_code C 0008 088A ADD.B R0L,R2L ;R2L + R0L -> R2L
24 BCD_code C 000A 0C1E MOV.W #H'0000,R2 ;Clear R2
25 BCD_code C 000C 5506 BSR TRANS
26 BCD_code C 0010 5502 BSR TRANS
27 BCD_code C 0012 5470 RTS
28                                      ;
29                                      ;
30                                      ;
31                                      ;
32 BCD_code C 0014 TRANS ;change BCD to hexadecimal
33 BCD_code C 0014 0CDE MOV.B R6L,R5L ;R6L -> R5L
34 BCD_code C 0016 7700 SLD #R,R0H ;load bit 0 of R0H
35 BCD_code C 0018 4406 BCC LBL2 ;Branch if C=0
36 BCD_code C 001A LBL1 MOV.W R5L,R6L ;R5L -> R6L
37 BCD_code C 001A 0CDE ADD.B #H'0F,R6L ;Clear bit 7-4 of R6L
38 BCD_code C 001E 4008 BRA LBL3 ;Branch always
39 BCD_code C 0020 LBL2 MOV.B R6L,R6L ;Shift REL 4 bit left
40 BCD_code C 0020 110E SHLR.B R6L ;Shift REL 4 bit left
41 BCD_code C 0022 110E SHLR.B R6L
42 BCD_code C 0024 110E SHLR.B R6L
43 BCD_code C 0026 110E SHLR.B R6L
44 BCD_code C 0028 LBL3 MOV.B R6L ;Shift Hexadecimal 1 bit left
45 BCD_code C 0028 100A SHLR.B R2L ;Shift Hexadecimal 1 bit left
46 BCD_code C 002A 1202 SOLT.B R2H
47 BCD_code C 002C 0D23 MOV.W R2,R3 ;R2 -> R3
48 BCD_code C 002E 100A SOLT.B R2L ;Shift Hexadecimal 2 bit left
49 BCD_code C 0030 1202 SOLT.B R2H
50 BCD_code C 0032 100A SOLT.B R2L
51 BCD_code C 0034 1202 SOLT.B R2H
52 BCD_code C 0036 0321 ADD.X R3,R2 ;R3 + R2 -> R2
53 BCD_code C 0038 0B6A ADD.B R6L,R6L
54 BCD_code C 003A 8200 ADDX.B #R,R2H
55 BCD_code C 003C 1A0S DEC.B #R,R0H ;Decrement bit counter
56 BCD_code C 003E 7700 SLD #R,R0H ;load bit 0 of R0H
57 BCD_code C 0040 4508 BCS LBL1 ;Branch if C=1
58 BCD_code C 0042 5470 RTS
59                                      ;
60                                      ;
61                                      ;
62                                      ;
63                                      ;
64                                      ;
65                                      ;
66                                      ;
67                                      ;
68                                      ;
69                                      ;
70                                      ;
71                                      ;
72                                      ;
73                                      ;
74                                      ;
75                                      ;
76                                      ;
77                                      ;
78                                      ;
79                                      ;
80                                      ;
81                                      ;
82                                      ;
83                                      ;
84                                      ;
85                                      ;
86                                      ;
87                                      ;
88                                      ;
89                                      ;
90                                      ;
91                                      ;
92                                      ;
93                                      ;
94                                      ;
95                                      ;
96                                      ;
97                                      ;
98                                      ;
99                                      ;
100                                     ;
101                                     ;
102                                     ;
103                                     ;
104                                     ;
105                                     ;
106                                     ;
107                                     ;
108                                     ;
109                                     ;
110                                     ;
111                                     ;
112                                     ;
113                                     ;
114                                     ;
115                                     ;
116                                     ;
117                                     ;
118                                     ;
119                                     ;
120                                     ;
121                                     ;
122                                     ;
123                                     ;
124                                     ;
125                                     ;
126                                     ;
127                                     ;
128                                     ;
129                                     ;
130                                     ;
131                                     ;
132                                     ;
133                                     ;
134                                     ;
135                                     ;
136                                     ;
137                                     ;
138                                     ;
139                                     ;
140                                     ;
141                                     ;
142                                     ;
143                                     ;
144                                     ;
145                                     ;
146                                     ;
147                                     ;
148                                     ;
149                                     ;
150                                     ;
151                                     ;
152                                     ;
153                                     ;
154                                     ;
155                                     ;
156                                     ;
157                                     ;
158                                     ;
159                                     ;
160                                     ;
161                                     ;
162                                     ;
163                                     ;
164                                     ;
165                                     ;
166                                     ;
167                                     ;
168                                     ;
169                                     ;
170                                     ;
171                                     ;
172                                     ;
173                                     ;
174                                     ;
175                                     ;
176                                     ;
177                                     ;
178                                     ;
179                                     ;
180                                     ;
181                                     ;
182                                     ;
183                                     ;
184                                     ;
185                                     ;
186                                     ;
187                                     ;
188                                     ;
189                                     ;
190                                     ;
191                                     ;
192                                     ;
193                                     ;
194                                     ;
195                                     ;
196                                     ;
197                                     ;
198                                     ;
199                                     ;
200                                     ;
201                                     ;
202                                     ;
203                                     ;
204                                     ;
205                                     ;
206                                     ;
207                                     ;
208                                     ;
209                                     ;
210                                     ;
211                                     ;
212                                     ;
213                                     ;
214                                     ;
215                                     ;
216                                     ;
217                                     ;
218                                     ;
219                                     ;
220                                     ;
221                                     ;
222                                     ;
223                                     ;
224                                     ;
225                                     ;
226                                     ;
227                                     ;
228                                     ;
229                                     ;
230                                     ;
231                                     ;
232                                     ;
233                                     ;
234                                     ;
235                                     ;
236                                     ;
237                                     ;
238                                     ;
239                                     ;
240                                     ;
241                                     ;
242                                     ;
243                                     ;
244                                     ;
245                                     ;
246                                     ;
247                                     ;
248                                     ;
249                                     ;
250                                     ;
251                                     ;
252                                     ;
253                                     ;
254                                     ;
255                                     ;
256                                     ;
257                                     ;
258                                     ;
259                                     ;
260                                     ;
261                                     ;
262                                     ;
263                                     ;
264                                     ;
265                                     ;
266                                     ;
267                                     ;
268                                     ;
269                                     ;
270                                     ;
271                                     ;
272                                     ;
273                                     ;
274                                     ;
275                                     ;
276                                     ;
277                                     ;
278                                     ;
279                                     ;
```
Section 9  Sorting

9.1  Set Constants

MCU:  H8/300 Series
      H8/300L Series

Label name:  SORT

9.1.1  Function

1. The software SORT sorts the data placed on the data memory, byte by byte, in descending order.
2. The number of bytes to be sorted can be up to 255.
3. Data to be sorted is represented as unsigned integers.

9.1.2  Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Number of bytes of data to be sorted</td>
<td>R0L</td>
</tr>
<tr>
<td></td>
<td>Start address of the data to be sorted</td>
<td>R4</td>
</tr>
<tr>
<td>Output</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

9.1.3  Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>*</td>
<td>*</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>U</th>
<th>H</th>
<th>U</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>×</td>
<td>*</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

×  : Unchanged
*  : Indeterminate
†  : Result
9.1.4 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>34</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>789482</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

9.1.5 Note

The clock cycle count (789482) in the specifications is for sorting 255-byte data in descending order.

9.1.6 Description

1. Details of functions
   a. The following arguments are used with the software SORT:
      R0L: Contains the number of bytes of data to be sorted – 1 as an input argument.
      R4: Contains the start address of the data to be sorted (stored on RAM).
b. Figure 9.1 shows an example of the software SORT being executed. When the input argument is set as shown in (1), the data is sorted in descending order as shown in (2).

Figure 9.1 Example of Software SORT Execution

2. Notes on usage
   a. Do not set "0" in R0L; otherwise, the software SORT will not operate normally.
   b. R0L must contain the number of bytes of data to be sorted - 1.

3. Data memory
   The software SORT does not use the data memory.
4. Example of use
Set the input arguments in registers and call the software SORT as a subroutine.

<table>
<thead>
<tr>
<th>WORK1</th>
<th>RES. B 1</th>
<th>Reserves a data memory area in which the user program places the number of bytes of data to be sorted.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORK2</td>
<td>RES. B 255</td>
<td>Reserves a data memory area in which the user program places the data to be sorted.</td>
</tr>
<tr>
<td>MOV. B</td>
<td>@WORK1, R0L</td>
<td>Places in the input argument the number of bytes of data to be sorted set by the user</td>
</tr>
<tr>
<td>MOV. W</td>
<td>#WORK2 R4</td>
<td>Places in R4 the start address of the data to be sorted set by the user program.</td>
</tr>
<tr>
<td>JSR</td>
<td>@SORT</td>
<td>Calls the software SORT as a subroutine.</td>
</tr>
</tbody>
</table>

5. Operation
a. Figure 9.2 shows an example of sorting where three pieces of data are sorted in descending order.

<table>
<thead>
<tr>
<th>Input data</th>
<th>5</th>
<th>10</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st sorting</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compare count</td>
<td>n-1=2</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>2nd sorting</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compare count</td>
<td>n-1=2</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Note] 
- : indicates comparison.
- : indicates exchange of data.

**Figure 9.2 Example of Sorting**

(i) The software searches the three pieces of data for the biggest number and sorts it at the extreme left. (See (1), (2) and (3) of figure 9.2.)

(ii) Next, the software identifies the greater of the second and last numbers as counted from the left and places it at the second place from the left. (See (4) and (5) of figure 9.2.)
b. Processing by programs

(i) R4 is used as the pointer for placing the biggest number. R5 is used as the pointer that indicates the address of the memory area containing the source number.

(ii) The comparand is placed in R1L.

(iii) The source number is placed in R1H.

(iv) R1L and R1H are compared with each other. If the source number is greater than the comparand (R1H > R1L), the two numbers are exchanged.

(v) The process (iii) to (iv) is repeated until the counter R0L indicating the remaining source numbers reaches "0".

(vi) When R0L reaches "0", the data stored in @R4 is assumed the biggest of the data compared.

(vii) The R0H indicating the number of remaining comparands is decremented.
9.1.7 Flowchart

- Sets R0L in R0H and R4 in R5.
- Stores the comparand in R1L.
- Increments R5.
- Stores the source number in R1H.
- Compares R1L with R1H. Branches if R1L ≥ R1H.
- Exchanges the comparand and the source number with each other if R1L < R1H.
- Decrements the counter indicating the remaining source numbers.
- Checks whether all source numbers have been compared.
- Increments R4.
- Increments R5.
- Stores the comparand in R1L.
- Stores the source number in R1H.
- Compares R1L with R1H. Branches if R1L ≥ R1H.
- Exchanges the comparand and the source number with each other if R1L < R1H.
- Decrements the counter indicating the remaining source numbers.
- Checks whether all source numbers have been compared.
- Decrements the counter indicating the remaining comparands.
- Check whether all comparands have been used.
- Increments R4.
9.1.8 Program List

*** H8/300 ASSEMBLER                VER 1.0B **   08/18/92 10:26:21

PROGRAM NAME =

1                                      ;********************************************************************
2                                      ;*
3                                      ;* 00 - NAME :SORTING (SORT)
4                                      ;*
5                                      ;********************************************************************
6                                      ;*
7                                      ;* ENTRY :ROL (BYTE NUMBER)
8                                      ;* R4 (START ADDRESS OF DATA)
9                                      ;*
10                                     ;* RETURN :NOTHING
11                                     ;*
12                                     ;********************************************************************
13                                     ;*
14 SORT_cod C 0000 .SECTION SORT_code,CODE,ALIGN=2
15                                      .EXPORT SORT
16                                      ;*
17 SORT_cod C 00000000 $                ;Entry point
18 SORT_cod C 0000 CB95 MOV.B R0L,R0H ;Set data counter
19 SORT_cod C 0002 0045 MOV.W R4,R5   ;R4 -> R5
20 SORT_cod C 0004 8849 MOV.B #84,R1L ;84 -> datal
21 SORT_cod C 0006 LBL1
22 SORT_cod C 0006 0803 ADDS.W @#1,R5 ;Increment address pointer1 (R5++)
23 SORT_cod C 0008 8851 MOV.B R5,R1H ;R5 -> data2
24 SORT_cod C 000A 1C19 CMP.B R1H,R1L
25 SORT_cod C 000C 4404 BHS LBL2 ;Branch if C=0
26 SORT_cod C 000E 8803 MOV.B R1L,R1H ;Store data1 @R5
27 SORT_cod C 0010 68C9 MOV.B #84,R4   ;4 -> datal2
28 SORT_cod C 0012 LBL2
29 SORT_cod C 0012 1A00 DEC.B R0H ;Decrement data counter
30 SORT_cod C 0014 4F03 BNE LBL1 ;Branch if Z=0
31 SORT_cod C 0016 88C9 MOV.B #84,R4   ;4 -> datal
32 SORT_cod C 0018 1A08 DEC.B R0L ;Decrement byte number
33 SORT_cod C 001A 4704 BEQ EXIT ;Branch Z=0
34 SORT_cod C 001C 0004 ADDS.W @#1,R4 ;Increment address pointer2 (R4++)
35 SORT_cod C 001E 4E00 BHA SORT ;Branch always
36 SORT_cod C 0020 EXIT
37 SORT_cod C 0020 5470 RTS
38                                      ;*
39                                      .END

*****TOTAL ERRORS       0
*****TOTAL WARNINGS     0
Section 10  ARRAY

10.1  2-Dimensional Array (ARRAY)

MCU  H8/300 Series
      H8/300L Series

Label name:  ARRAY

10.1.1  Function

1. The software ARRAY retrieves data from a 2-dimensional array (hereafter called an "array") and sets its address and elements (x, y) of the array when the data matches.
2. Data to be processed by the software ARRAY are 1-byte unsigned integers.
3. The elements of an array are 1-byte unsigned integers.
4. An array can be set up in the range 255 bytes × 255 bytes.

10.1.2  Arguments

<table>
<thead>
<tr>
<th>Description</th>
<th>Memory area</th>
<th>Data length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Data to be retrieved</td>
<td>R0L</td>
<td>1</td>
</tr>
<tr>
<td>Start address of the array</td>
<td>R4</td>
<td>2</td>
</tr>
<tr>
<td>Number of rows of the array</td>
<td>R2L</td>
<td>1</td>
</tr>
<tr>
<td>Number of columns of the array</td>
<td>R3L</td>
<td>1</td>
</tr>
<tr>
<td>Output Address of matched data</td>
<td>R4</td>
<td>2</td>
</tr>
<tr>
<td>Array element x of matched data</td>
<td>R5H</td>
<td>1</td>
</tr>
<tr>
<td>Array element y of matched data</td>
<td>R5L</td>
<td>1</td>
</tr>
<tr>
<td>Presence of matched data</td>
<td>C flag (CCR)</td>
<td></td>
</tr>
</tbody>
</table>

10.1.3  Internal Register and Flag Changes

<table>
<thead>
<tr>
<th>R0H</th>
<th>R0L</th>
<th>R1</th>
<th>R2H</th>
<th>R2L</th>
<th>R3H</th>
<th>R3L</th>
<th>R4</th>
<th>R5H</th>
<th>R5L</th>
<th>R6</th>
<th>R7</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>
10.1.4 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory (bytes)</td>
<td>46</td>
</tr>
<tr>
<td>Data memory (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Stack (bytes)</td>
<td>0</td>
</tr>
<tr>
<td>Clock cycle count</td>
<td>1986</td>
</tr>
<tr>
<td>Reentrant</td>
<td>Possible</td>
</tr>
<tr>
<td>Relocation</td>
<td>Possible</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Possible</td>
</tr>
</tbody>
</table>

10.1.5 Note

The clock cycle count (1986) in the specifications is for the example shown in figure 10.1. If either element x or y is "0", the software terminates immediately and clears the C flag.
10.1.6 Description

1. Details of functions
   a. The following arguments are used with the software ARRAY:
      (i) Input arguments
         - R0L: Data to be retrieved
         - R4: Start address of the array
         - R2L: Number of rows of the array (x)
         - R3L: Number of columns of the array (y)
      (ii) Output arguments
         - R4: Address of the matched data
         - R5H: Array element x of the matched data
         - R5L: Array element y of the matched data
         - C flag (CCR): Indicates the state at the end of the software ARRAY.
           - C flag = 1: Matched data is found on the array.
           - C flag = 0: Matched data is not found on the array.
   b. Figure 10.1 shows an example of the software ARRAY being executed. When the input arguments are set as shown in (1), the software ARRAY references the array (16 × 16) in figure 10.2 and places the address of the matched data in R4, the array element x in R5H, and the array element y in R5L as shown in (2).
Figure 10.1  Example of Software ARRAY Execution

Figure 10.2  Array Space

c. Execution of the software ARRAY requires an array as shown in figure 10.3.
(i) The size of an array is identified by the number of rows \((x)\) and the number of columns \((y)\).

(ii) The elements of an array are represented by \(x\) (row) and \(y\) (column), which range from \((0, 0)\) to \((x - 1, y - 1)\).

(iii) The start address of an array is \((0, 0)\), at which retrieval starts in the order as shown in figure 10.3.

2. Notes on usage
   Do not set "0" as \(x\) and \(y\); otherwise, the software ARRAY do not start retrieval of data, clears the C flag, and terminates.

3. Data Memory
   The software ARRAY does not use the data memory.

4. Example of Use
   Set the data to be retrieved and the start address, the number of rows and the number of columns of an array to be searched, and call the software ARRAY as a subroutine.
Reserves a data memory area for the start address of the array.

Reserves a data memory area for the number of rows of the array (x).

Reserves a data memory area for the number of columns of the array (y).

Reserves a data memory area for the data to be retrieved.

Reserves a data memory area for the address of the matched data.

Reserves a data memory area for the element (x) of the array when the data is matched.

Reserves a data memory area for the element (y) of the array when the data is matched.

Places the data to be retrieved.

Places the start address of the array.

Places the number of rows of the array (x).

Places the number of columns of the array (y).

Calls the software ARRAY as a subroutine.

Stores the address of the matched data.

Stores the element of the array (x) when the data is matched.

Stores the element of the array (y) when the data is matched.
10.1.7 Flowchart

```
ARRAY

R4 → R1  #H'0000 → R6

YES R2L = R6L
NO

YES R3L = R6L
NO

R2L × R3L → R3

LBL1 @R4 → R0H

YES R0L = R0H
NO

YES R3 - #1 → R3

NO R3 = R6

R4 + #1 → R4

LBL2 R4 → R5

R5 - R1 → R5

R5 - R2L → R5

1 → Cfrag

EXIT1 0 → Cfrag

EXIT2 RTS

- Sets R4 in R1 and clears R6.
- Exits if the number of rows or columns of the array is "0".
- Finds the size of the array.
- Stores the array data sequentially in R0H and branches if the data is matched.
- Exits if no matched data exists in the array data.
- Finds the elements of the array by doing division.
- Indicates that the matched data existed in the array data. (C flag = "1")
- Indicates that the matched data did not exist in the array data or the size of the array was "0". (C flag = "0")

... Exits if the number of rows or columns of the array is "0".
... Finds the size of the array.
... Stores the array data sequentially in R0H and branches if the data is matched.
... Exits if no matched data exists in the array data.
... Finds the elements of the array by doing division.
... Indicates that the matched data existed in the array data. (C flag = "1")
... Indicates that the matched data did not exist in the array data or the size of the array was "0". (C flag = "0")
```
### 10.1.8 Program List

```
1 ;********************************************************************
2 ;* 00 - NAME : 2-DIMENSIONAL ARRAY (ARRAY)
3 ;********************************************************************
4 ;*
5 ;********************************************************************
6 ;*
7 ;* ENTRY : R0L (REFERENCE DATA)
8 ;* R2L (NUMBER OF COLUMN [X])
9 ;* R3L (NUMBER OF ROW  [Y])
10 ;*
11 ;* RETURNS : R5M (ARRAY ELEMENT OF COLUMN [x])
12 ;* R5L (ARRAY ELEMENT OF LOW  [y])
13 ;*
14 ;* C flag OF CCR (C=1;TRUE , C=0;FALSE)
15 ;*
16 ;*
17 ;********************************************************************
18 ;
19 ARRAY_co C 0000 .SECTION ARRAY_code,CODE,ALIGN=2
20 .EXPORT ARRAY
21 ;
22 ARRAY_co C 00000000 ARRAY .EQU $  ;Entry point
23 ARRAY_co C 0000 0041 MOV.W R4,R1
24 ARRAY_co C 0002 79060000 MOV.W #H'0000,R6  ;Clear R6
25 ARRAY_co C 0006 1CAE MULXU R2L,R3L ;Get total number of array(R3)
26 ARRAY_co C 0010 LBL1
27 ARRAY_co C 0010 6840 MOV.B @R4,R0H ;Load array data
28 ARRAY_co C 0012 1C80 CMP.B R0L,R0H ;Branch if data find
29 ARRAY_co C 0014 1B03 SUBS.W #1,R3 ;Decrement R3
30 ARRAY_co C 0016 51A5 DIVXU R2L,R5L ;Get array element [x,y]
31 ARRAY_co C 0018 0401 ORC.B #H'01,CCR ;Set C flag of CCR
32 ARRAY_co C 001A 4002 BRA LBL1 ;Branch always
33 ARRAY_co C 0020 LBL2
34 ARRAY_co C 0020 0D45 MOV.W R4,R5
35 ARRAY_co C 0022 1915 SUB.W R1,R5 ;Clear C flag of CCR
36 ARRAY_co C 0024 3A5A DIV.C R2L,R5 ;Set array element [x,y]
37 ARRAY_co C 0026 0401 DIVC.B R6',CCR ;Set C flag of CCR
38 ARRAY_co C 0028 4002 BRA EXIT1 ;Branch always
39 ARRAY_co C 002A EXIT1
40 ARRAY_co C 002A 06FE ANDC.B #H'FE,CCR ;Clear C flag of CCR
41 ARRAY_co C 002C EXIT2
42 ARRAY_co C 002C 5470 RTS
```

---

*** H8/300 ASSEMBLER                VER 1.0B **   08/18/92 10:26:53
PROGRAM NAME =

```
296
```

---

***** TOTAL ERRORS     0
***** TOTAL WARNINGS   0